COMPUTER SCIENCE

A Structured Programming Approach Using C

Behrouz A. Forouzan & Richard F. Gilberg

THIRD EDITION

Operator	Description	Example	*	Assoc	Pr

	Identifiers
	Constants
	Parenthetical Expressions
		`amount`
		`3.14159`
		`(a + b)`
			N	N/A	16

`[ ]`	Array Index	`ary[i]`	N	Left-Right	16
`f(...)`	Function Call	`doIt(x, y)`	Y		
`.`	Direct Member Selection	`str.mem`	N		
`->`	Indirect Member Selection	`ptr->mem`	N		
`++ --`	Postfix Increment • Decrement	`a++`	Y		
`++ --`	Prefix Increment • Decrement	`++a`	Y	Right-Left	15
`sizeof`	Size in Bytes	`sizeof(int)`	N		
`~`	Ones Complement	`~a`	N		
`!`	Not	`!a`	N		
`+ -`	Plus • Minus	`+a`	N		
`&`	Address	`&a`	N		
`*`	Dereference / Indirection	`*ptr`	N		
`( )`	Type Cast	`(int)ptr`	N	Right-Left	14
`* / %`	Multiply • Divide • Modulus	`a * b`	N	Left-Right	13
`+ -`	Addition • Subtraction	`a + b`	N	Left-Right	12
`<< >>`	Bit Shift Left • Bit Shift Right	`a << 3`	N	Left-Right	11
`< <= > >=`	Comparison	`a < 5`	N	Left-Right	10
`== !=`	Equal • Not Equal	`a == b`	N	Left-Right	9
`&`	Bitwise And	`a & b`	N	Left-Right	8
`^`	Bitwise Exclusive Or	`a ^ b`	N	Left-Right	7
`|`	Bitwise Or	`a | b`	N	Left-Right	6
`&&`	Logical And	`a && b`	N	Left-Right	5
`||`	Logical Or	`a || b`	N	Left-Right	4
`? :`	Conditional	`a ? x : y`	N	Right-Left	3

`= += -=`
`*= /= %=`
`>>= <<=`
`&= ^= |=`
	Assignment	`a = 5`
		`a %= b`
		`a &= c`
		`a |= d`
			Y	Right-Left	2

`,`	Comma	`a, b, c`	N	Left-Right	1

* Side Effects (Yes / No)

Precedence Table

x++ : Add 1 to x and return old value
++x : Add 1 to x and return new value

Computer Science: A Structured Programming Approach Using C, Third Edition

Australia • Brazil • Japan • Korea • Mexico • Singapore • Spain • United Kingdom • United States

Preface to Third Edition

This text has two primary objectives: to teach the basic principles of programming as outlined in the ACM curriculum for a CS1 class and to teach the basic constructs of the C Language. Our text puts these objectives in the context of good software engineering concepts that we have developed through more than 30 years of experience in industry and academia.

Major Changes in Third Edition

This is a major change from the second edition. In addition to significant changes in the presentation and material covered, it also updates the language to the ISO/IEC 9899 (1999), or as it is more commonly known, C99.

While we have made major changes, we have held true to our primary objectives as outlined above. The text remains a comprehensive introduction to computer programming in a software engineering context.

The major changes are outlined in the following sections.

Standard C (1999)

The following topics, in alphabetical order, are explicitly discussed in the text.

- the Boolean type
- character set extensions (including Unicode and wide-characters)
- complex arithmetic
- extended math library
- *for* statement declarations
- integer type extensions
- line comments

Extended Chapter Material

In addition to the changes required by the inclusion of the new C99 extensions, several chapters have been revised or reorganized.

- A new section in Chapter 4, "Inter-Function Communication," provides a better understanding of how functions communicate.

- The concept of "incremental program development" is presented through an example that starts in Chapter 4 and is extended in Chapter 5. A second example in Chapter 8 completes the discussion.
- The discussion of text and binary files has been reworked to present the materials in a more logical, and less redundant, fashion.
- The presentation of structured types is changed to conform to C99 and common usage.
- Chapter 14, "Bitwise Operators," has been extended and includes more examples. In particular, we added several applications related to the Internet and network programming.
- Chapter 15, "Lists," has also been extended. It now includes an overview of the basic elements of data structures, including introductory discussions of stack, queues, lists, trees, and graphs.
- Additional flowcharts and structure charts are included throughout the text to provide better design considerations for some of the more complex examples.
- Appendix D, "Numbering Systems," provides an expanded presentation of the material.
- Appendix I, "Pointers to *void* and to Functions," has been expanded to include a discussion of using pointer to *void* types.
- Appendix K. "Program Development," is a new discussion of program development concepts.
- Appendix L, "Understanding Complex Declarations," contains the discussion on reading complex declarations, which has been removed from Chapter 10.

A C Language Perspective

While C is a complex and professional language, our classroom experience using the previous editions of this book has shown that beginning students can easily understand it. We believe that if the language is put into a perspective that allows the student to understand its design, C is not a difficult language.

There are two aspects of C that separate it from most other languages: expressions and pointers. The concept of expressions, as used in C, is unique. The first half of this text builds a firm understanding of expressions. We introduce pointers only to the extent necessary to cover passing them to functions in Chapter 4 and arrays in Chapter 8. Our experiences have shown that with a firm grasp of the expression concept, much of the mystery of C disappears.

In Chapters 9, we begin to develop the concept of pointers, which are further developed in Chapter 10. Chapter 11 provides a discussion of C's approach to strings. Finally, Chapter 14 provides an extensive discussion of bitwise operators, a subject that is needed to make the book complete.

The last chapter is a simple introduction to data structures. While not all courses will have time to cover this chapter, those that do will give the students a head start into data structures. Motivated students will find that they can "get a leg up" on data structures over the term break.

The appendices at the end of the text comprise a rich set of references to subjects required in the complete C language. These will be of use to students who go on to take other courses in the C language.

Features of the Book

Several features of this book make it unique and easy for beginning students to understand.

Structure and Style

One of our basic tenets is that good habits are formed early. The corollary is that bad habits are hard to break. Therefore, we consistently emphasize the principles of structured programming and software engineering. Throughout each chapter, short code snippets and programs are used to demonstrate techniques.

Complete programs, generally found in the last section of the chapter, are written in a well-document consistent style. As programs are analyzed, style and standards are further explained. While we acknowledge that there are many good styles, our experience has shown that if students are exposed to one good style and adopt it, they will be better able to adapt to other good styles.

Principle Before Practice

Whenever possible, we develop the principle of a subject before we introduce the language implementation. For example, in Chapter 5 we first introduce the concept of logical data and selection and then we introduce the *if . . . else* and *switch* statements. This approach gives the student an understanding of selection before introducing the nuances of the language.

Visual Approach

A brief scan of the book will demonstrate that our approach is visual. There are more than 400 figures, plus more than 70 tables and 180 program examples. While this amount of material tends to create a large book, the visual approach makes it easy for students to follow the material.

Examples

While the programming examples vary in complexity, each uses a consistent style. Our experience working with productional programs that live for 10 to 20 years convinced us that readable and understandable programs are easier to work with than programs written in a terse, cryptic manner. For that reason, and to emphasize the structure of the language, we label the sections in a function with comments. We consistently follow a style that places only one declaration, definition, or statement on a line.

The source code for these programs is available on the Course Technology Web site (see "Web Site Support Materials" later in this Preface). Students are able to run these programs and to explore related topics through modification and experimentation.

Coding Techniques

Throughout the text we include coding techniques that make programs more readable and often more efficient. For example, in the analylsis of Program 5-4 you will find the following discussion:

> Where do we check for *greater than*? The answer is that we default the *greater than* condition to the outer *else*. . . . When coding a two-way selection statement, try to code the most probable condition first; with nested selection statements, code the most probable first and the least probable last.

These techniques are drawn from our extensive industry and classroom experience.

Software Engineering

A discussion of software engineering principles concludes each chapter. Our intent is not to replace a separate course in software engineering. Rather, we firmly believe that by incorporating basic software engineering principles early in their studies, students will be better prepared for a formal treatment of the subject. Even more important, by writing well-engineered programs from the beginning, students will not be forced to unlearn and relearn. They will better understand software discussions in their subsequent classes.

While the software engineering sections are found at the end of each chapter, they are most successfully taught by introducing them as the chapter unfolds. Then, a short review at the end of the chapter summarizes the principles that have been demonstrated during the lectures.

These sections are visually distinguishable from the rest of the chapter. They have been set apart for several reasons. First, they are in reality a small book within a book. While these sections contain important material, the book stands on its own without them. You may, therefore, decide to cover the software engineering sections as formal lecture topics or informally while the chapter material is being covered. You may decide to assign them to the student as additional reading, or, you may decide to exclude them entirely from the course.

In general, software engineering sections directly or indirectly pertain to the chapter material. Where they don't, they discuss general software engineering subjects, such as cohesion, coupling, and quality.

Pedagogical Material

End chapter material meets two pedagogical objectives: first, it helps the students to review or summarize what they have learned, and second, it tests the students' mastery of the chapter material.

Review Material

- **Tips and Common Programming Errors** points out helpful hints and possible problem areas.
- **Key Terms** provides a list of the important terms introduced in the chapter.
- **Summary** contains a concise overview of the key points for students to understand in the chapter.

Practice Sets

- **Review Questions** contain several multiple choice questions that are similar to the questions found in the examination database. The answers to the odd-numbered questions are included in the solutions on the Course Technology web site.
- **Exercises** are short questions covering the material in the chapter. The answers to the odd-numbered exercises are also included in the solutions on the Course Technology web site.
- **Problems** are short coding problems, generally intended to be run on a computer. They can usually be developed in two to three hours. Once again, odd-numbered solutions are found on the web site.
- **Projects** are longer, major assignments that may take the average student six to nine hours to develop.

Appendices and Cover Material

The appendices are intended to provide quick reference material, such as the Unicode Character Set, or to provide a review of material, such as numbering systems, usually covered in a general computer class.

The inside covers contain two important references that are used continually throughout the course—the Precedence Table and the Formatted I/O Codes.

Web Site Support Materials

Included on the Course Technology Web site (http://www.course.com) are two sets of materials, one for the professor and one for the student.

Professor Materials

The professor's materials include the solutions to all review questions, exercises, and problems. Because the projects are more complex assignments for which there is no standard answer, we do not provide solutions for them.

In addition to the end-material solutions, there are three other sets of support materials: (1) Copies of all programs in the text. The programs have been extensively tested on at least two different computers with different operating systems. All of them conform to the C99 Standard.

(2) PowerPoint materials covering objectives, figures, programs, important points, and chapter summaries. The PowerPoint materials can be used as is or edited to suit individual class needs. This flexibility allows professors to argument the text materials with their own. It also allows them to rearrange the materials to suit their individual needs and style.

(3) New with the third edition is the availability of ExamView®. This objective-based test generator lets the professor create paper, LAN, or Web-based tests from testbanks designed specifically for this text. Using the QuickTest Wizard, they can easily and quickly create true/false and multiple-choice tests. It is also possible to add questions that cover supplemental material provided by the professor.

Student Materials

The student materials include the solutions to the odd-numbered review questions, exercises, and problems. They also have access to the copies of the programs used in the text

Acknowledgments

No book of this scope can be developed without the support of many people.

Reviewers

To anyone who has not been through the process, the value of peer reviews cannot be appreciated enough. Writing a text rapidly becomes a myopic process. The important guidance of reviewers who can stand back and review the text as a whole cannot be measured. We would especially like to acknowledge the contributions of the reviewers of all three editions.

Stephen Allen, *Utah State University*
Mary Astone, *Troy State University*
Ali Behforooz, *Towson State University*
George Berry, *Wentworth Institute of Technology*
Ernest Carey, *Utah Valley State College*
Ping Chu Chu, *Fayetteville State University*

Constance Conner, *City College of San Francisco*
John S. DaPonte, *Southern Connecticut State University*
Maurice L. Eggen, *Trinity University*
Peter Gabrovsky, *CSU Northridge*
Robert Gann, *Hartwick College*
Henry Gordon, *Kutztown University*
Rick Graziani, *Cabrillo College*
Barbara Guillott, *Louisiana State University*
Jerzy Jaromczyk, *University of Kentucky*
John Kinio, *Humber College*
Roberta Klibaner, *College of Staten Island*
Joseph A. Konstan, *University of Minnesota*
Krishna Kulkarni, *Rust College*
John Lowther, *Michigan Technological University*
Mike Michaelson, *Palomar College*
Kara Nance, *University of Alaska—Fairbanks*
Ali Nikzad, *Huston-Tillotson College*
Jo Ann Parikh, *Southern Connecticut State University*
Mark Parker, *Shoreline Community College*
Savitha Pinnepalli, *Louisiana State University*
Oskar Reiksts, *Kutztown University*
Jim Roberts, *Carnegie Mellon University*
Ali Salenia, *South Dakota State University*
Larry Sells, *Oklahoma City University*
Shashi Shekhar, *University of Minnesota*
Robert Signorile, *Boston College*
Brenda Sonderegger, *Montana State University*
Deborah Sturm, *College of Staten Island*
Venkat Subramanian, *University of Houston*
John B. Tappen, *University of Southern Colorado*
Marc Thomasm, *California State University, Bakersfield*
John Trono, *St. Michael's College*
K.C. Wong, *Fayetteville State University*

Course Technology Staff

Our thanks to our editors, Alyssa Pratt, Senior Product Manager, and Mary Franz, Senior Acquisitions Editor, for helping us produce the book. We are also indebted to the Quality Assurance staff who diligently double-checked each chapter and program. Our thanks to Burt LaFountain, Serge Palladino, and Chris Scriver.

We would also like to thank the staff at GEX Publishing Services, especially Sandra Mitchell, who ably guided the book through production.

Behrouz A. Forouzan
Richard F. Gilberg

Contents

Chapter 2 Introduction to the C Language 29

Chapter 3 Structure of a C Program 93

Chapter 1

Introduction to Computers

Welcome to computer science! You are about to explore a wonderful and exciting world—a world that offers many challenging and exciting careers.

In this chapter, we introduce you to the concepts of computer science, especially as they pertain to computer programming. You will study the concept of a computer system and how it relates to computer hardware and software. We will also present a short history of computer programming languages so that you understand how they have evolved and how the C language fits into the picture.

We will then describe how to write a program, first with a review of the tools and steps involved, and then with a review of a system development methodology.

Objectives

- ❑ To review basic computer systems concepts
- ❑ To be able to understand the different computing environments and their components
- ❑ To review the history of computer languages
- ❑ To be able to list and describe the classifications of computer languages
- ❑ To understand the steps in the development of a computer program
- ❑ To review the system development life cycle

1.1 Computer Systems

Today computer systems are found everywhere. Computers have become almost as common as televisions. But what is a computer? A computer is a system made of two major components: hardware and software. The computer hardware is the physical equipment. The software is the collection of programs (instructions) that allow the hardware to do its job. Figure 1-1 represents a **computer system**.

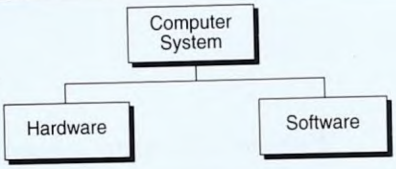


FIGURE 1-1 A Computer System

Computer Hardware

The **hardware** component of the computer system consists of five parts: input devices, central processing unit (CPU), primary storage, output devices, and auxiliary storage devices (Figure 1-2).

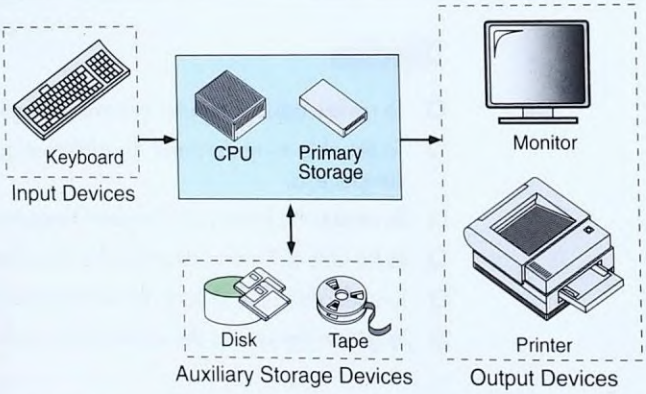


FIGURE 1-2 Basic Hardware Components

The **input device** is usually a keyboard where programs and data are entered into the computer. Examples of other input devices include a mouse, a pen or stylus, a touch screen, or an audio input unit.

The **central processing unit (CPU)** is responsible for executing instructions such as arithmetic calculations, comparisons among data, and movement of data inside the system. Today's computers may have one, two, or more CPUs. **Primary storage**, also known as **main memory**, is a place where the programs and data are stored temporarily during processing. The data in primary storage are erased when we turn off a personal computer or when we log off from a time-sharing computer.

The **output device** is usually a monitor or a printer to show output. If the output is shown on the monitor, we say we have a **soft copy**. If it is printed on the printer, we say we have a **hard copy**.

Auxiliary storage, also known as **secondary storage**, is used for both input and output. It is the place where the programs and data are stored permanently. When we turn off the computer, our programs and data remain in the secondary storage, ready for the next time we need them.

Computer Software

Computer **software** is divided into two broad categories: system software and application software. This is true regardless of the hardware system architecture. System software manages the computer resources. It provides the interface between the hardware and the users but does nothing to directly serve the users' needs. Application software, on the other hand, is directly responsible for helping users solve their problems. Figure 1-3 shows this breakdown of computer software.

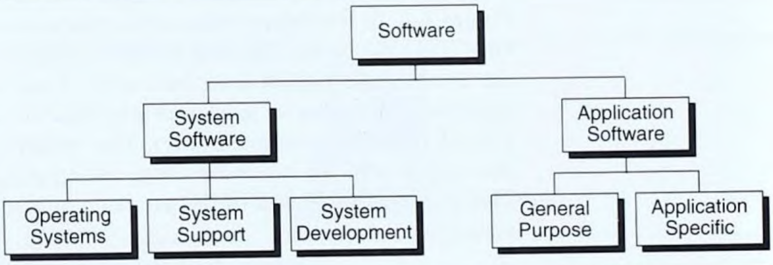


FIGURE 1-3 Types of Software

System Software

System software consists of programs that manage the hardware resources of a computer and perform required information processing tasks. These programs are divided into three classes: the operating system, system support, and system development.

The **operating system** provides services such as a user interface, file and database access, and interfaces to communication systems such as Internet protocols. The primary purpose of this software is to keep the system operating in an efficient manner while allowing the users access to the system.

System support software provides system utilities and other operating services. Examples of system utilities are sort programs and disk format programs. Operating services consist of programs that provide performance statistics for the operational staff and security monitors to protect the system and data.

The last system software category, **system development software**, includes the language translators that convert programs into machine language for execution, debugging tools to ensure that the programs are error-free, and computer-assisted software engineering (CASE) systems that are beyond the scope of this book.

Application Software

Application software is broken into two classes: general-purpose software and application-specific software. **General-purpose software** is purchased from a software developer and can be used for more than one application. Examples of general-purpose software include word processors, database management systems, and computer-aided design systems. They are labeled general purpose because they can solve a variety of user computing problems.

Application-specific software can be used only for its intended purpose. A general ledger system used by accountants and a material requirements planning system used by a manufacturing organization are examples of application-specific software. They can be used only for the task for which they were designed; they cannot be used for other generalized tasks.

The relationship between system and application software is seen in Figure 1-4. In this figure, each circle represents an interface point. The inner core is the hardware. The user is represented by the outer layer. To work with the system, the typical user uses some form of application software. The application software in turn interacts with the operating system, which is a part of the system software layer. The system software provides the direct interaction with the hardware. Note the opening at the bottom of the figure. This is the path followed by the user who interacts directly with the operating system when necessary.

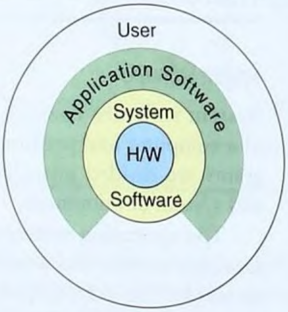


FIGURE 1-4 Relationship Between System and Application Software

If users cannot buy software that supports their needs, then a custom-developed application must be built. In today's computing environment, one of the tools used to develop software is the C language that we will be studying in this text.

1.2 Computing Environments

In the early days of computers, there was only one environment: the mainframe computer hidden in a central computing department. With the advent of minicomputers and personal computers, the environment changed, resulting in computers on virtually every desktop. In this section we describe several different environments.

Personal Computing Environment

In 1971, Marcian E. Hoff, working for Intel, combined the basic elements of the central processing unit into the microprocessor. This first computer on a chip was the Intel 4004 and was the grandparent many times removed of Intel's current system.

If we are using a personal computer, all of the computer hardware components are tied together in our **personal computer** (or **PC**[1] for short). In this situation, we have the whole computer for ourself; we can do whatever we want. A typical personal computer is shown in Figure 1-5.

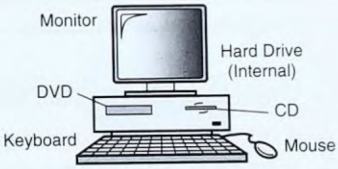


FIGURE 1-5 Personal Computing Environment

Time-Sharing Environment

Employees in large companies often work in what is known as a **time-sharing environment**. In the time-sharing environment, many users are connected to one or more computers. These computers may be minicomputers or central mainframes. The terminals they use are often nonprogrammable, although today we see more and more microcomputers being used to simulate terminals. Also, in the time-sharing environment, the output devices (such as printers) and auxiliary storage devices (such as disks) are shared by all of the

[1] PC is now generally accepted to mean any hardware using one of Microsoft's Windows operating systems, as opposed to Apple's Macintosh. We use it in the original, generic sense meaning any personal computer.

users. A typical college lab in which a minicomputer is shared by many students is shown in Figure 1-6.

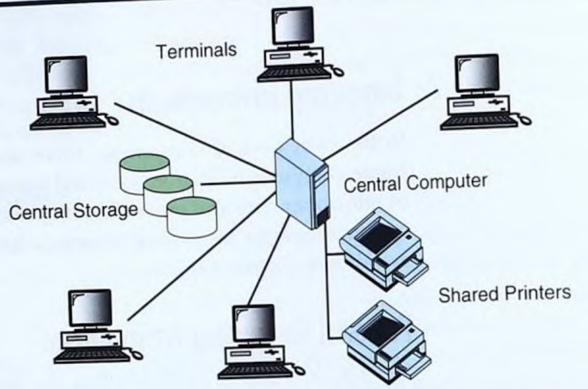


FIGURE 1-6 Time-sharing Environment

In the time-sharing environment, all computing must be done by the central computer. In other words, the central computer has many duties: It must control the shared resources; it must manage the shared data and printing; and it must do the computing. All of this work tends to keep the computer busy. In fact, it is sometimes so busy that the user becomes frustrated by the computer's slow responses.

Client/Server Environment

A **client/server** computing environment splits the computing function between a central computer and users' computers. The users are given personal computers or workstations so that some of the computation responsibility can be moved from the central computer and assigned to the workstations. In the client/server environment, the users' microcomputers or workstations are called the **client**. The central computer, which may be a powerful microcomputer, minicomputer, or central mainframe system, is known as the **server**. Because the work is now shared between the users' computers and the central computer, response time and monitor display are faster and the users are more productive. Figure 1-7 shows a typical client/server environment.

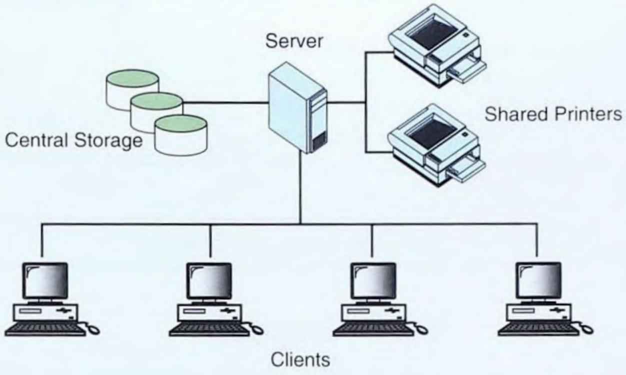


FIGURE 1-7 The Client/Server Environment

Distributed Computing

A **distributed computing** environment provides a seamless integration of computing functions between different servers and clients. The Internet provides connectivity to different servers throughout the world. For example, eBay uses several computers to provide its auction service. This environment provides a reliable, scalable, and highly available network. Figure 1-8 shows a distributed computing system.

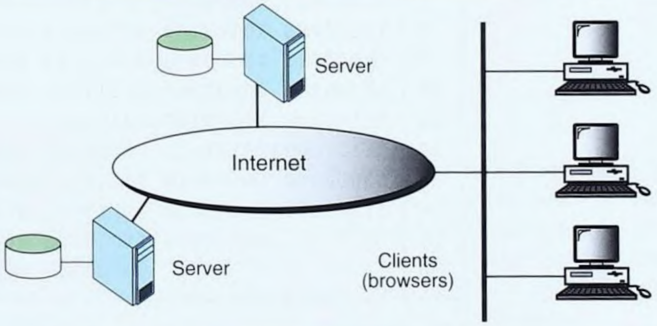


FIGURE 1-8 Distributed Computing

1.3 Computer Languages

To write a program for a computer, we must use a **computer language**. Over the years computer languages have evolved from machine languages to natural languages. A summary of computer languages is seen in Figure 1-9.

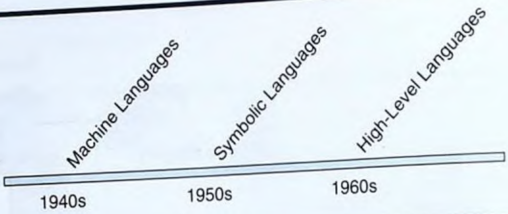


FIGURE 1-9 Computer Language Evolution

Machine Languages

In the earliest days of computers, the only programming languages available were machine languages. Each computer has its own **machine language**, which is made of streams of 0's and 1's. Program 1-1 shows an example of a machine language. This program multiplies two numbers and prints the results.

PROGRAM 1-1 The Multiplication Program in Machine Language

```
         00000000 00000100 0000000000000000
01011110 00001100 11000010 0000000000000010
         11101111 00010110 0000000000000101
         11101111 10011110 0000000000001011
11111000 10101101 11011111 0000000000010010
         01100010 11011111 0000000000010101
11101111 00000010 11111011 0000000000010111
11110100 10101101 11011111 0000000000011110
00000011 10100010 11011111 0000000000100001
11101111 00000010 11111011 0000000000100100
01111110 11110100 10101101
11111000 10101110 11000101 0000000000101011
00000110 10100010 11111011 0000000000110001
11101111 00000010 11111011 0000000000110100
         01010000 11010100 0000000000111011
                  00000100 0000000000111101
```

The instructions in machine language must be in streams of 0's and 1's because the internal circuits of a computer are made of switches, transistors, and other electronic devices that can be in one of two states: off or on. The off state is represented by 0; the on state is represented by 1.

The only language understood by computer hardware is machine language.

Symbolic Languages

It became obvious that few programs would be written if programmers continued to work in machine language. In the early 1950s, Admiral Grace Hopper, a mathematician and naval officer, developed the concept of a special computer program that would convert programs into machine language. These early programming languages simply mirrored the machine languages using symbols, or mnemonics, to represent the various machine language instructions. Because they used symbols, these languages were known as **symbolic languages**. Program 1-2 shows the multiplication program in a symbolic language.[2]

PROGRAM 1-2 The Multiplication Program in Symbolic Language

```
        entry   main,^m<r2>
        subl2   #12,sp
        jsb     C$MAIN_ARGS
        movab   $CHAR_STRING_CON

        pushal  -8(fp)
        pushal  (r2)
        calls   #2,SCANF
        pushal  -12(fp)
        pushal  3(r2)
        calls   #2,SCANF
        mull3   -8(fp),-12(fp),-
        pusha   6(r2)
        calls   #2,PRINTF
        clrl    r0
        ret
```

Because a computer does not understand symbolic language, it must be translated to the machine language. A special program called an assembler translates symbolic code into machine language. Because symbolic languages had to be assembled into machine language, they soon became known as **assembly languages**. This name is still used today for symbolic languages that closely represent the machine language of their computer.

Symbolic language uses symbols, or mnemonics, to represent the various machine language instructions.

[2] The symbolic language format is label, operators, and operands. There are no labels in this example, so the left column is empty.

High-Level Languages

Although symbolic languages greatly improved programming efficiency, they still required programmers to concentrate on the hardware that they were using. Working with symbolic languages was also very tedious because each machine instruction had to be individually coded. The desire to improve programmer efficiency and to change the focus from the computer to the problem being solved led to the development of **high-level languages**.

High-level languages are portable to many different computers, allowing the programmer to concentrate on the application problem at hand rather than the intricacies of the computer. High-level languages are designed to relieve the programmer from the details of the assembly language. High-level languages share one thing with symbolic languages, however: They must be converted to machine language. The process of converting them is known as **compilation**.

The first widely used high-level language, FORTRAN,[3] was created by John Backus and an IBM team in 1957; it is still widely used today in scientific and engineering applications. Following soon after FORTRAN was COBOL.[4] Admiral Hopper was again a key figure in the development of the COBOL business language.

C, with its standard libraries, is considered a high-level language used for system software and new application code. Program 1-3 shows the multiplication program as it would appear in the C language.

PROGRAM 1-3 The Multiplication Program in C

```
/* This program reads two integers from the keyboard
   and prints their product.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Definitions
   int number1;
   int number2;
   int result;

// Statements
```

continued

[3] FORTRAN is an acronym for FORmula TRANslation.

[4] COBOL is an acronym for COmmon Business-Oriented Language.

PROGRAM 1-3 The Multiplication Program in C *(continued)*

```
    scanf  ("%d", &number1);
    scanf  ("%d", &number2);
    result = number1 * number2;
    printf ("%d", result);
    return 0;
 }  // main
```

1.4 Creating and Running Programs

As we learned in the previous section, computer hardware understands a program only if it is coded in its machine language. In this section, we explain the procedure for turning a program written in C into machine language. The process is presented in a straightforward, linear fashion, but you should recognize that these steps are repeated many times during development to correct errors and make improvements to the code.

It is the job of the programmer to write and test the program. There are four steps in this process: (1) writing and editing the program, (2) compiling the program, (3) linking the program with the required library modules, and (4) executing the program. These steps are seen in Figure 1-10.

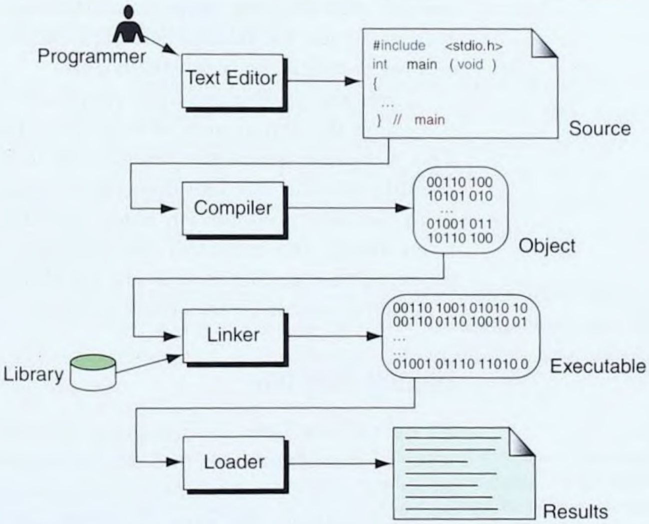


FIGURE 1-10 Building a C Program

Writing and Editing Programs

The software used to write programs is known as a **text editor**. A text editor helps us enter, change, and store character data. Depending on the editor on our system, we could use it to write letters, create reports, or write programs. The main difference between text processing and program writing is that programs are written using lines of code, while most text processing is done with characters and lines.

Our text editor could be a generalized word processor, but it is more often a special editor included with the compiler. Some of the features you should look for in your editor are search commands to locate and replace statements, copy and paste commands to copy or move statements from one part of a program to another, and formatting commands that allow you to set tabs to align statements.

After we complete a program, we save our file to disk. This file will be input to the compiler; it is known as a **source file**.

Compiling Programs

The code in a source file stored on the disk must be translated into machine language. This is the job of the **compiler**. The C compiler is actually two separate programs: the **preprocessor** and the **translator**.

The preprocessor reads the source code and prepares it for the translator. While preparing the code, it scans for special instructions known as **preprocessor commands**. These commands tell the preprocessor to look for special code libraries, make substitutions in the code, and in other ways prepare the code for translation into machine language. The result of preprocessing is called the **translation unit**.

After the preprocessor has prepared the code for compilation, the translator does the actual work of converting the program into machine language. The translator reads the translation unit and writes the resulting **object module** to a file that can then be combined with other precompiled units to form the final program. An object module is the code in machine language. Even though the output of the compiler is machine language code, it is not yet ready to run; that is, it is not yet executable because it does not have the required C and other functions included.

Linking Programs

As we will see later, a C program is made up of many functions. We write some of these functions, and they are a part of our source program. However, there are other functions, such as input/output processes and mathematical library functions, that exist elsewhere and must be attached to our program. The **linker** assembles all of these functions, ours and the system's, into our final executable program.

Executing Programs

Once our program has been linked, it is ready for execution. To execute a program, we use an operating system command, such as `run`, to load the program into primary memory and execute it. Getting the program into memory is the function of an operating system program known as the **loader**. It locates the executable program and reads it into memory. When everything is loaded, the program takes control and it begins execution. In today's integrated development environments, these steps are combined under one mouse click or pull-down window.

In a typical program execution, the program reads data for processing, either from the user or from a file. After the program processes the data, it prepares the output. Data output can be to the user's monitor or to a file. When the program has finished its job, it tells the operating system, which then removes the program from memory.

1.5 System Development

We've now seen the steps that are necessary to build a program. In this section, we discuss *how* we go about developing a program. This critical process determines the overall quality and success of our program. If we carefully design each program using good structured development techniques, our programs will be efficient, error-free,[5] and easy to maintain.

System Development Life Cycle

Today's large-scale, modern programming projects are built using a series of interrelated phases commonly referred to as the **system development life cycle**. Although the exact number and names of the phases differ depending on the environment, there is general agreement as to the steps that must be followed. Whatever the methodology, however, today's software engineering concepts require a rigorous and systematic approach to software development.[6]

One very popular development life cycle is the **waterfall model**. Depending on the company and the type of software being developed, this model consists of between five and seven phases. Figure 1-11 is one possible variation on the model.

[5] Many computer scientists believe that all programs contain at least one *bug*—an undetected error—that is just waiting to cause problems, given the right set of circumstances. Programs have run for years without problems only to fail when an unusual situation occurs. Perhaps the most famous bug was the one known as Y2K because it caused programs to fail on January 1, 2000.

[6] For a discussion of various models, see *Software Engineering: A Practitioner's Approach,* 5th. ed. by Roger S. Pressman, (New York, NY: McGraw-Hill, 2001).

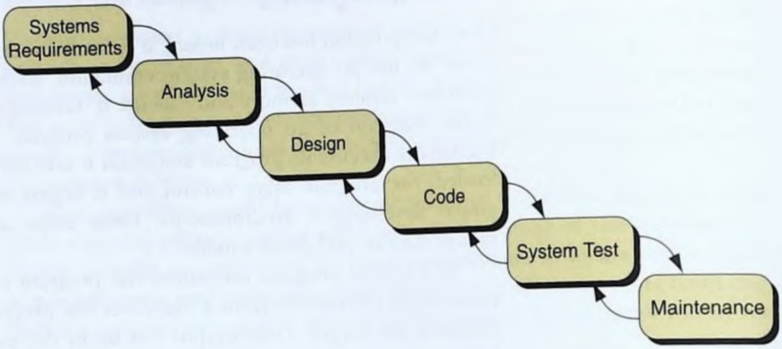


FIGURE 1-11 Waterfall Model

The waterfall model starts with *systems requirements*. In this phase, the systems analyst defines requirements that specify what the proposed system is to accomplish. The requirements are usually stated in terms that the user understands. The *analysis* phase looks at different alternatives from a system's point of view, while the *design* phase determines how the system will be built. In the design phase, the functions of the individual programs that will make up the system are determined and the design of the files and/or the databases is completed. Finally, in the fourth phase, *code*, we write the programs. This is the phase that is explained in this book. After the programs have been written and tested to the programmer's satisfaction, the project proceeds to *system test*. All of the programs are tested together to make sure the system works as a whole. The final phase, *maintenance*, keeps the system working once it has been put into production.

Although the implication of the waterfall approach is that the phases flow in a continuous stream from the first to the last, this is not really the case. Note the iteration as indicated by the backward-flowing arrows in Figure 1-11. As each phase is developed, errors and omissions will often be found in the previous work. When this happens, it is necessary to go back to the previous phase to rework it for consistency and to analyze the impact caused by the changes. Hopefully, this is a short rework. We are aware of at least three major projects, however, that were in the code and test phases when it was determined that they could not be implemented and had to be canceled. When this happens, millions of dollars and years of development are lost.

Program Development

Program Development is a multistep process that requires that we understand the problem, develop a solution, write the program, and then test it.

When we are given the assignment to develop a program, we will be given a program requirements statement and the design of any program interfaces. We should also receive an overview of the complete project so that we will understand how our part fits in to the whole. Our job is to determine how to take the inputs we are given and convert them into the outputs that have been specified. This is known as *program design.* To give us an idea of how this process works, let's look at a simple problem: Calculate the square footage of a house. How do we go about doing this?

Understand the Problem

The first step in solving any problem is to understand it. We begin by reading the requirements statement carefully. When we think that we fully understand it, we review our understanding with the user and the systems analyst. Often this involves asking questions to confirm our understanding.

For example, after reading our simple requirements statement, we should ask several clarifying questions.

- ❑ What is the definition of square footage?
- ❑ How is the square footage going to be used?
    - ■ for insurance purposes?
    - ■ to paint the inside or outside of the house?
    - ■ to carpet the whole house?
- ❑ Is the garage included?
- ❑ Are closets and hallways included?

Each of these potential uses requires a different measure. If we don't clarify the exact purpose—that is, if we make assumptions about how the output is going to be used—we could supply the wrong answer.

As this example shows, even the simplest problem statements need clarification. Imagine how many questions must be asked for a programmer to write a program that will contain hundreds or thousands of detailed statements.

Develop the Solution

Once we fully understand the problem and have clarified any questions we may have, we need to develop our solution. Three tools will help us in this task: (1) structure charts, (2) pseudocode, and (3) flowcharts. Generally, we will use only two of them—a structure chart and either pseudocode or a flowchart.

The structure chart is used to design the whole program. Pseudocode and flowcharts, on the other hand, are used to design the individual parts of the program. These parts are known as modules in pseudocode or functions in the C language.

Structure Chart

A **structure chart**, also known as a hierarchy chart, shows the functional flow through our program. Large programs are complex structures consisting of many interrelated parts; thus, they must be carefully laid out. This task is similar to that of a design engineer who is responsible for the operational design of any complex item. The major difference between the design built by a programmer and the design built by an engineer is that the programmer's product is software that exists only inside the computer, whereas the engineer's product is something that can be seen and touched.

The structure chart shows how we are going to break our program into logical steps; each step will be a separate module. The structure chart shows the interaction between all the parts (modules) of our program.

It is important to realize that the design, as represented by the structure chart, is done before we write our program. In this respect, it is like the architect's blueprint. We would not start to build a house without a detailed set of plans. Yet one of the most common errors of both experienced and new programmers alike is to start coding a program before the design is complete and fully documented.

This rush to start is due in part to programmers' thinking they fully understand the problem and also due to their excitement about solving a new problem. In the first case, what they find is that they did not fully understand the problem. By taking the time to design the program, they will raise more questions that must be answered and therefore will gain a better understanding of the problem.

An old programming proverb: Resist the temptation to code.

The second reason programmers code before completing the design is just human nature. Programming is a tremendously exciting task. To see our design begin to take shape, to see our program creation working for the first time, brings a form of personal satisfaction that is a natural high.

In the business world, when we complete a structure chart design, we will convene a review panel for a *structured walk-through* of our program. Such a panel usually consists of a representative from the user community, one or two peer programmers, the systems analyst, and possibly a representative from the testing organization. In the review, we will walk our review team through our structure chart to show how we plan to solve the objectives of our program. The team will then offer constructive suggestions as to how to improve our design.

The primary intent of the review is to increase quality and save time. The earlier a mistake is detected, the easier it is to fix it. If we can eliminate only one or two problems with the structured walk-through, the time will be well spent. Naturally, in a programming class, you will not be able to convene a full panel and conduct a formal walk-through. What you can do, however, is review your design with some of your classmates and with your professor.

Looking again at our problem to calculate the square footage of a house, let's assume the following answers to the questions raised in the previous section.

1. The purpose of calculating the square footage is to install new floor covering.
2. Only the living space will be carpeted. The garage and closets will not be considered.
3. The kitchen and bathrooms will be covered with linoleum; the rest of the house is to be carpeted.

With this understanding, we decide to write separate modules for the kitchen, bathroom(s), bedrooms, family room, and living room. We use separate modules because the various rooms may require a different quality of linoleum and carpeting. The structure chart for our design is shown in Figure 1-12.

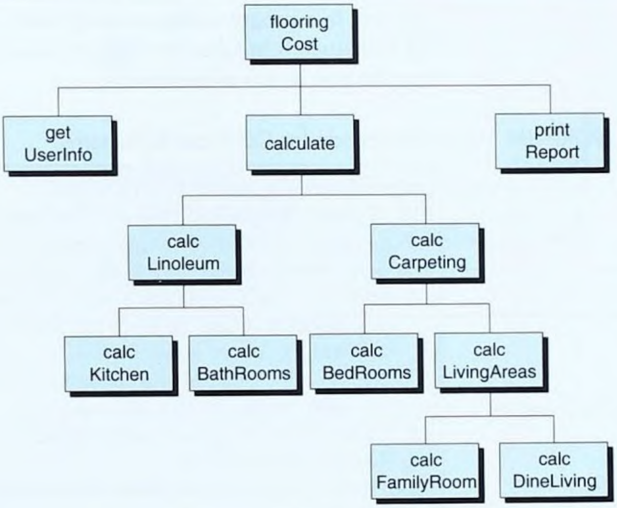


FIGURE 1-12 Structure Chart for Calculating Square Footage

Whether you use a flowchart or pseudocode to complete the design of your program will depend on your experience, the difficulty of the process you are designing, and the culture and standards of the organization where you are working. We believe that new programmers should first learn program design by flowcharting because a flowchart is a visual tool that is easier to create than pseudocode. On the other hand, pseudocode is more common among professional programmers.

Pseudocode

Pseudocode is part English, part program logic. Its purpose is to describe, in precise algorithmic detail, what the program being designed is to do. This requires defining the steps to accomplish the task in sufficient detail so that they can be converted into a computer program. Pseudocode excels at this type of precise logic. The pseudocode for determining the linoleum for the bathroom is shown in Algorithm 1-1.

Pseudocode

English-like statements that follow a loosely defined syntax and are used to convey the design of an algorithm.

Most of the statements in the pseudocode are easy to understand. A prompt is simply a displayed message telling the user what data are to be entered. The *while* is a loop that repeats the three statements that follow it and uses the number of bathrooms read in statement 2 to tell when to stop. Looping is a programming concept that allows us to repeat a block of code. We will study it in Chapter 6. In this case, it allows us to process the information for one or more bathrooms.

ALGORITHM 1-1 Pseudocode for Calculate Bathrooms

```
Algorithm Calculate BathRooms
 1 prompt user and read linoleum price
 2 prompt user and read number of bathrooms
 3 set total bath area and baths processed to zero
 4 while ( baths processed < number of bathrooms )
   1  prompt user and read bath length and width
   2  total bath area =
   3           total bath area + bath length * bath width
   4  add 1 to baths processed
 5 bath cost = total bath area * linoleum price
 6 return bath cost
end Algorithm Calculate BathRooms
```

Flowchart

A **flowchart** is a program design tool in which standard graphical symbols are used to represent the logical flow of data through a function. Appendix C contains complete instructions for creating flowcharts. If you are not familiar with flowcharts, we suggest you read Appendix C now.

The flowchart in Figure 1-13 shows the design for calculating the area and cost for the bathrooms. A few points merit comment here. This flowchart is basically the same as the pseudocode. We begin with prompts for the price of the linoleum and the number of bathrooms and read these two pieces of data. (As a general rule, flowcharts do not explicitly show standard concepts such as prompts.) The loop reads the dimensions for each bathroom. Finally, when we know the total area, we calculate the price and return to `calcLinoleum`.

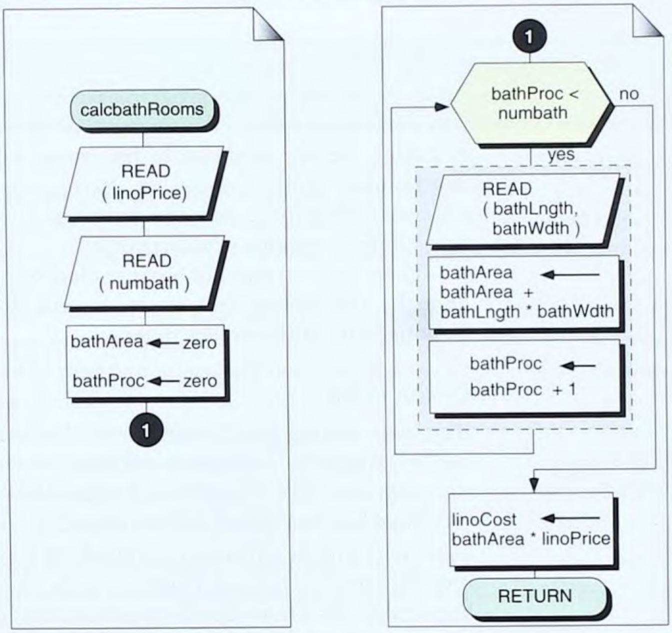


FIGURE 1-13 Flowchart for Calculate Bathrooms

Write the Program

Now it's time to write the program! But first, let's review the steps that we've used.

1. Understand the problem.
2. Develop a solution.
   a. Design the program—create the structure chart.
   b. Design the algorithms for the program using either flowcharting or pseudocode or both.
3. Write the program.

When we write a program, we start with the top box on the structure chart and work our way to the bottom. This is known as top–down implementation. You will find that it is a very easy and natural way to write programs, especially if you have done a solid job on your design.

For our first few programs, there will be only one module, representing the top box of the structure chart. The first programs are quite simple and do not require subdivision. When we get to Chapter 4, however, we will begin to write functions and our structure charts will get larger. At that time we will point out some more techniques for writing structured programs. In the meantime, concentrate on writing good pseudocode or flowcharts for the main part of our programs.

Test the Program

After we've written our program, we must test it. Program testing can be a very tedious and time-consuming part of program development. As the programmer, we are responsible for completely testing our program. In large development projects, there are often specialists known as test engineers who are responsible for testing the system as a whole—that is, for testing to make sure all the programs work together.

There are two types of testing: blackbox and whitebox. Blackbox testing is done by the system test engineer and the user. Whitebox testing is the responsibility of the programmer.

Blackbox Testing

Blackbox testing gets its name from the concept of testing the program without knowing what is inside it—without knowing how it works. In other words, the program is like a black box that we can't see into.

Blackbox test plans are developed by looking only at the requirements statement (this is only one reason why it is so important to have a good set of requirements). The test engineer uses these requirements and his or her knowledge of systems development and the user working environment to create a test plan that will then be used when the system is tested as a whole. We should ask to see this test plan before we write our program. The test engineer's plan will help us make sure we fully understand the requirements and also help us create our own test plan.

Whitebox Testing

Whereas blackbox testing assumes that the tester knows nothing about the program, **whitebox testing** assumes that the tester knows everything about the program. In this case, the program is like a glass house in which everything is visible.

Whitebox testing is our responsibility. As the programmer, we know exactly what is going on inside the program. We must make sure that every instruction and every possible situation has been tested. That is not a simple task!

Experience will help us design good test data, but one thing we can do from the start is to get in the habit of writing test plans. We should start the test plan when we are in the design stage. As we build your structure chart, ask ourself what situations, especially unusual situations, we need to test for and make a note of them immediately because we won't remember them an hour later.

When we are writing our flowcharts or pseudocode, we need to review them with an eye toward test cases and make additional notes of the cases we may need. Finally, while we are coding, we need to keep paper handy (or a test document open in our word processor) to make notes about test cases we need.

Except for the most simple program, one set of test data will not completely validate a program.

When it is time to construct our test cases, we review our notes and organize them into logical sets. Except for very simple student programs, one set of test data will never completely validate a program. For large-scale development projects, 20, 30, or even more test cases may need to be run to validate a program.

Finally, while we are testing, we will think of more test cases. Again, write them down and incorporate them into our test plan. After our program is finished and in production, we will need the test plan again when we make modifications to the program.

How do we know when our program is completely tested? In reality, there is no way to know for sure. But there are a few things we can do to help the odds. While some of these concepts will not be clear until you have read other chapters, we include them here for completeness.

1. Verify that every line of code has been executed at least once. Fortunately, there are programming tools on the market today that will do this for us.
2. Verify that every conditional statement in our program has executed both the true and false branches, even if one of them is null. (See Chapter 5.)
3. For every condition that has a range, make sure the tests include the first and last items in the range, as well as items below the first and above the last—the most common mistakes in array range tests occur at the extremes of the range. (See Chapter 8.)
4. If error conditions are being checked, make sure all error logic is tested. This may require us to make temporary modifications to our program to force the errors. (For instance, an input/output error usually cannot be created—it must be simulated.)

1.6 Software Engineering

Software engineering is the establishment and use of sound engineering methods and principles to obtain software that is reliable and that works on real machines.[7] This definition, from the first international conference on software engineering in 1969, was proposed 30 years after the first computer was built. During that period, software was more of an art than a science. In fact, one of the most authoritative treatments of programming describes it as an art: *The Art of Computer Programming*. This three-volume series, originally written by Donald E. Knuth in the late 1960s and early 1970s,[8] is considered the most complete discussion of many computer science concepts.

Because the science and engineering base for building reliable software did not yet exist, programs written in the 1950s and 1960s were a maze of complexity known as "spaghetti code." It was not until Edsger Dijkstra wrote a letter to the editor of the *Communications of the ACM* (Association of Computing Machinery)[9] in 1968 that the concept of structured programming began to emerge.

Dijkstra was working to develop algorithms that would mathematically prove program accuracy. He proposed that any program could be written with only three constructs or types of instructions: (1) sequences, (2) the *if...else* selection statement, and (3) the *while* loop. As we will see, language developers have added constructs, such as the *for* loop and the *switch* in C. These additional statements are simply enhancements to Dijkstra's basic constructs that make programming easier. Today, virtually all programming languages offer structured programming capabilities.

Throughout this text we will be emphasizing the concepts of good software engineering. Chief among them is the concept of structured programming and a sound programming style. A section in each chapter will include a discussion of these concepts with specific emphasis on the application of the material in the chapter.

The tools of programming design have also changed over the years. In the first generation of programming, one primary tool was a block diagram. This tool provided boxes, diamonds, and other flowchart symbols to represent different instructions in a program. Each instruction was contained in a separate symbol. This concept allowed programmers to write a program on paper and check its logic flow before they entered it in the computer.

With the advance of symbolic programming, the block diagram gave way to the flowchart. Although the block diagram and flowchart look similar, the flowchart does not contain the detail of the block diagram. Many instructions are implied by the descriptive names put into the boxes; for example, the read

[7] F. L Bauer, Technical University, Munich, Germany (1969).

[8] Donald E. Knuth, *The Art of Computer Programming,* vols. 1 (Third Edition, 1997), 2 (Third Edition, 1997), and 3 (Second Edition, 1998). (Reading, Mass.: Addison-Wesley, 1977).

[9] Edsger W. Dijkstra, "Go To Statement Considered Harmful," *Communications of the ACM*, vol. 11, no. 3 (March 1968).

statements in Figure 1-13 imply the prompt. Flowcharts have largely given way to other techniques in program design, but they are still used today by many programmers for working on a difficult logic problem.

Today's programmers are most likely to use a high-level design tool such as tight English or pseudocode. We will use pseudocode throughout the text to describe many of the algorithms we will be developing.

Finally, the last several years have seen the automation of programming through the use of computer-assisted software engineering (CASE) tools. These tools make it possible to determine requirements, design software, and develop and test software in an automated environment using programming workstations. The discussion of the CASE environment is beyond the scope of this text and is left for courses in systems engineering.

1.7 Tips and Common Errors

1. Become familiar with the text editor in your system so you will be able to create and edit your programs efficiently. The time spent learning different techniques and shortcuts in a text editor will save time in the future.
2. Also, become familiar with the compiler commands and keyboard shortcuts. On most computers, a variety of options are available to be used with the compiler. Make yourself familiar with all of these options.
3. Read the compiler's error messages. Becoming familiar with the types of error messages and their meanings will be a big help as you learn C.
4. Remember to save and compile your program each time you make changes or corrections in your source file. When your program has been saved, you won't lose your changes if a program error causes the system to fail during testing.
5. Run your program many times with different sets of data to be sure it does what you want.
6. The most common programming error is not following the old proverb to "resist the urge to code." Make sure you understand the requirements and take the time to design a solution before you start writing code.

1.8 Key Terms

application software
application-specific software
assembly language
auxiliary storage
blackbox testing
central processing unit (CPU)
client
client/server
compilation
compiler
computer language
computer system
distributed environment
executable program
flowchart
general-purpose software
hard copy
hardware
high-level language
input device
linker
operating system
output device
personal computer (PC)
preprocessor
preprocessor commands
primary storage
program development
pseudocode
secondary storage
server
soft copy
software
source file
structure chart
symbolic language
system development life cycle
system development software
system software
system support software
text editor
time-sharing environment

loader
machine language
main memory
object module
translation unit
translator
waterfall model
whitebox testing

1.9 Summary

□ A computer system consists of hardware and software.

□ Computer hardware consists of a central processing unit (CPU), primary memory, input devices, output devices, and auxiliary storage.

□ Software consists of two broad categories: system software and application software.

□ The components of system software are the operating system, system support, and system development.

□ Application software is divided into general-purpose applications and application-specific software.

□ Over the years, programming languages have evolved from machine language, to symbolic language, and to high-level languages.

□ The C language is a high-level language.

□ The software used to write programs is known as a text editor.

□ The file created from a text editor is known as a source file.

□ The code in a source file must be translated into machine language using the C compiler, which is made of two separate programs: the preprocessor and the translator.

□ The file created from the compiler is known as an object module.

□ An object module is linked to the standard functions necessary for running the program by the linker.

□ A linked program is run using a loader.

□ The system development life cycle is a series of interrelated steps that provide a rigorous and systematic approach to software development.

□ To develop a program, a programmer must complete the following steps:

a. Understand the problem.

b. Develop a solution using structure charts and either flowcharts or pseudocode.

c. Write the program.

d. Test the program.

□ The development of a test plan starts with the design of the program and continues through all steps in program development.

□ Blackbox testing consists primarily of testing based on user requirements.

- ❑ Whitebox testing, executed by the programmer, tests the program with full knowledge of its operational weaknesses.
- ❑ Testing is one of the most important parts of your programming task. You are responsible for whitebox testing; the systems analyst and user are responsible for blackbox testing.
- ❑ Software engineering is the application of sound engineering methods and principles to the design and development of application programs.

1.10 Practice Sets

Review Questions

1. Computer software is divided into two broad categories: system software and operational software.

   a. True
   b. False

2. The operating system provides services such as a user interface, file and database access, and interfaces to communications systems.

   a. True
   b. False

3. The first step in system development is to create a source program.

   a. True
   b. False

4. The programmer design tool used to design the whole program is the flowchart.

   a. True
   b. False

5. Blackbox testing gets its name from the concept that the program is being tested without knowing how it works.

   a. True
   b. False

6. Which of the following is a component(s) of a computer system?

   a. Hardware
   b. Software
   c. Both hardware and software
   d. Pseudocode
   e. System test

7. Which of the following is an example of application software?

   a. Database management system
   b. Language translator

- c. Operating system
- d. Sort
- e. Security monitor

8. Which of the following is not a computer language?
   - a. Assembly/symbolic language
   - b. Binary language
   - c. High-level languages
   - d. Machine language
   - e. Natural language

9. The computer language that most closely resembles machine language is
   - a. Assembly/symbolic
   - b. COBOL
   - c. FORTRAN
   - d. High-level

10. The tool used by a programmer to convert a source program to a machine language object module is a
    - a. Compiler
    - b. Language translator
    - c. Linker
    - d. Preprocessor
    - e. Text editor

11. The ________________ contains the programmer's original program code.
    - a. Application file
    - b. Executable file
    - c. Object file
    - d. Source file
    - e. Text file

12. The series of interrelated phases that is used to develop computer software is known as
    - a. Program development
    - b. Software engineering
    - c. System development life cycle
    - d. System analysis
    - e. System design

13. The ________________ is a program design tool that is a visual representation of the logic in a function within a program.
    - a. Flowchart
    - b. Program map
    - c. Pseudocode
    - d. Structure chart
    - e. Waterfall model

14. The test that validates a program by ensuring that all of its statements have been executed—that is, by knowing exactly how the program is written—is
    a. Blackbox testing
    b. Destructive testing
    c. Nondestructive testing
    d. System testing
    e. Whitebox testing

Exercises

15. Describe the two major components of a computer system.

16. Computer hardware is made up of five parts. List and describe them.

17. Describe the major differences between a time-sharing and a client/server environment.

18. Describe the two major categories of software.

19. What is the purpose of an operating system?

20. Identify at least two types of system software that you will use when you write programs.

21. Give at least one example of general-purpose and one example of application-specific software.

22. List the levels of computer languages discussed in the text.

23. What are the primary differences between symbolic and high-level languages?

24. What is the difference between a source program and an object module?

25. Describe the basic steps in the system development life cycle.

26. What documentation should a programmer receive to be able to write a program?

27. List and explain the steps that a programmer follows in writing a program.

28. Describe the three tools that a programmer may use to develop a program solution.

29. What is meant by the old programming proverb, "Resist the temptation to code"?

30. What is the difference between blackbox and whitebox testing?

31. What is software engineering?

Problems

32. Write pseudocode for *calcLivingAreas*, Figure 1-12, "Structure Chart for Calculating Square Footage."

33. Create a flowchart for a routine task, such as calling a friend, that you do on a regular basis.

34. Write pseudocode for the flowchart you created in Problem 33.

Chapter 2

Introduction to the C Language

In Chapter 1, we traced the evolution of computer languages from the machine languages to high-level languages. As mentioned, **C** (the language used exclusively in this book) is a high-level language. Since you are going to spend considerable time working with the language, you should have some idea of its origins and evolution.

In this chapter we introduce the basics of the C language. You will write your first program, which is traditionally known in C as the "Hello World," or "Greeting" program. Along the way we will introduce you to the concepts of data types, constants, and variables. Finally, you will see two C library functions that read and write data. Since this chapter is just an introduction to C, most of these topics are covered only in sufficient detail to enable you to write your first program. They will be fully developed in future chapters.

Objectives

- ❑ To understand the structure of a C-language program
- ❑ To write your first C program
- ❑ To introduce the *include* preprocessor command
- ❑ To be able to create good identifiers for objects in a program
- ❑ To be able to list, describe, and use the C basic data types
- ❑ To be able to create and use variables and constants in a program
- ❑ To understand input and output concepts as they apply to C programs
- ❑ To be able to use simple input and output statements
- ❑ To understand the software engineering role in documentation, data naming, and data hiding

2.1 Background

C is a structured programming language. It is considered a high-level language because it allows the programmer to concentrate on the problem at hand and not worry about the machine that the program will be using. While many languages claim to be machine independent, C is one of the closest to achieving that goal. That is another reason why it is used by software developers whose applications have to run on many different hardware platforms.

C, like most modern languages, is derived from ALGOL, the first language to use a block structure. ALGOL never gained wide acceptance in the United States, but it was widely used in Europe.

ALGOL's introduction in the early 1960s paved the way for the development of structured programming concepts. Some of the first work was done by two computer scientists, Corrado Bohm and Guiseppe Jacopini, who published a paper in 1966 that defined the concept of structured programming. Another computer scientist, Edsger Dijkstra, popularized the concept. His letter to the editors of the *Communications of the ACM* (Association of Computing Machinery) brought the structured programming concept to the attention of the computer science community.

Several obscure languages preceded the development of C. In 1967, Martin Richards developed a language he called Basic Combined Programming Language, or BCPL. Ken Thompson followed in 1970 with a similar language he simply called B. B was used to develop the first version of UNIX, one of the popular network operating systems in use today. Finally, in 1972, Dennis Ritchie developed C, which took many concepts from ALGOL, BCPL. This path, along with several others, is shown in Figure 2-1.

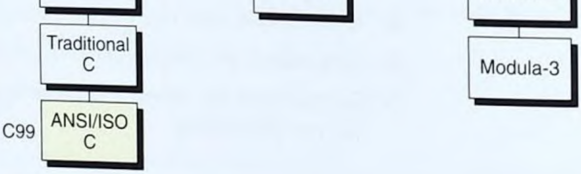


FIGURE 2-1 Taxonomy of the C Language

What is known as traditional C is this 1972 version of the language, as documented and popularized in a 1978 book by Brian W. Kernighan and Dennis Ritchie.[1] In 1983, the American National Standards Institute (ANSI) began the definition of a standard for C. It was approved in December 1989. In 1990, the International Standards Organization (ISO) adopted the ANSI standard. This version of C is known as C89.

In 1995, minor changes were made to the standard. This version is known as C95. A much more significant update was made in 1999. The changes incorporated into the standard, now known as C99, are summarized in the following list.

1. Extensions to the character type to support non-English characters
2. A Boolean type
3. Extensions to the integer type
4. Inclusion of type definitions in the *for* statement.
5. Addition of imaginary and complex types
6. Incorporation of the C++ style line comment (//)

We use the Standard C in this book.

2.2 C Programs

It's time to write your first C program! This section will take you through all the basic parts of a C program so that you will be able to write it.

Structure of a C Program

Every C program is made of one or more preprocessor commands, a global declaration section, and one or more functions. The **global declaration section** comes at the beginning of the program. We will talk more about it later, but the basic idea of global declarations is that they are visible to all parts of the program.

The work of the program is carried out by its functions, blocks of code that accomplish a task within a program. One, and only one, of the functions must be named *main*. The *main* function is the starting point for the program. All functions in a program, including *main*, are divided into two sections: the declaration section and the statement section. The **declaration section** is at the beginning of the function. It describes the data that you will

[1] Brian Kernighan and Dennis Ritchie, *The C Programming Language*, 2nd ed. (Englewood Cliffs, N.J.: Prentice Hall, 1989).

be using in the function. Declarations in a function are known as local declarations (as opposed to global declarations) because they are visible only to the function that contains them.

The **statement section** follows the declaration section. It contains the instructions to the computer that cause it to do something, such as add two numbers. In C, these instructions are written in the form of **statements**, which gives us the name for the section.

Figure 2-2 shows the parts of a simple C program. We have explained everything in this program but the preprocessor commands. They are special instructions to the preprocessor that tell it how to prepare the program for compilation. One of the most important of the preprocessor commands, and one that is used in virtually all programs, is **include**. The include command tells the preprocessor that we need information from selected libraries known as **header files**. In today's complex programming environments, it is almost impossible to write even the smallest of programs without at least one library function. In your first program, you will use one include command to tell C that you need the input/output library to write data to the monitor.

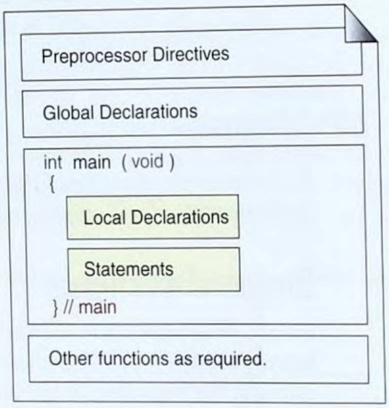
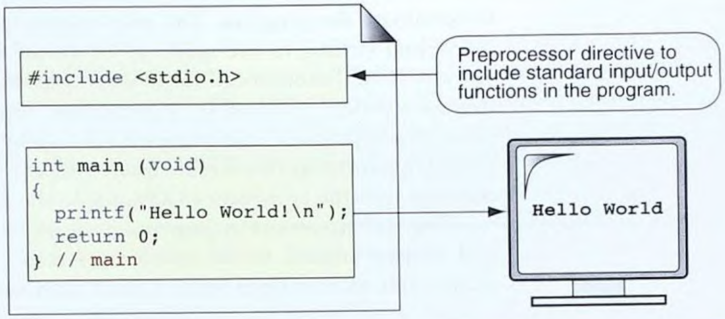


FIGURE 2-2 Structure of a C Program

Your First C Program

Your first C program will be very simple (see Figure 2-3). It will have only one preprocessor command, no global declarations, and no local definitions. Its purpose will be simply to print a greeting to the user. Therefore, its statement section will have only two statements: one that prints a greeting and one that stops the program.



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

```
int main (void)
{
  printf("Hello World!\n");
  return 0;
} // main
```

```
Hello World
```

Preprocessor directive to include standard input/output functions in the program.

FIGURE 2-3 The Greeting Program

Preprocessor Commands

The preprocessor commands come at the beginning of the program. All preprocessor commands start with a pound sign (#); this is just one of the rules of C known as its **syntax.** Preprocessor commands can start in any column, but they traditionally start in column 1.

The preprocessor command tells the compiler to include the standard input/output library file in the program. You need this library file to print a message to the terminal. Printing is one of the input/output processes identified in this library. The complete syntax for this command is shown below.

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

The syntax of this command must be exact. Since it is a preprocessor command, it starts with the pound sign. There can be no space between the pound sign and the keyword, *include*. Include means just what you would think it does. It tells the preprocessor that you want the library file in the pointed brackets (< >) *included* in your program. The name of the header file is *stdio.h*. This is an abbreviation for "standard input/output header file."

main

The executable part of your program begins with the function *main*, which is identified by the function header shown below. We explore the meaning of the function syntax in Chapter 4. For now, all you need to understand is that *int* says that the function will return an integer value to the operating system, that the function's name is *main*, and that it has no parameters (the parameter list is void). Note that there is no punctuation after the function header.

```
int main (void)
```

Within *main* there are two statements: one to print your message and one to terminate the program. The print statement uses a library function to do the actual writing to the monitor. To invoke or execute this print function, you *call* it. All function call statements consist of the name of the function, in this case *printf*, followed by a **parameter list** enclosed in parentheses. For your simple program, the parameter list simply contains what you want displayed, enclosed in two double quote marks ("..."). The \n at the end of the message tells the computer to advance to the next line in the output.

The last statement in your program, `return 0`, terminates the program and returns control to the operating system. One last thing: The function *main* starts with an open brace ({)and terminates with a close brace (}).

Comments

Although it is reasonable to expect that a good programmer should be able to read code, sometimes the meaning of a section of code is not entirely clear. This is especially true in C. Thus, it is helpful if the person who writes the code places some **comments** in the code to help the reader. Such comments are merely internal **program documentation**. The compiler ignores these comments when it translates the program into executable code. To identify a comment, C uses two different formats: block comments and line comments.

Block Comment

A **block comment** is used when the comment will span several lines. We call this comment format block comment. It uses opening and closing comment tokens. A **token** is one or more symbols understood by the compiler that help it interpret code. Each comment token is made of two characters that, taken together, form the token; there can be no space between them. The opening token is /* and the closing token is */. Everything between the opening and closing comment tokens is ignored by the compiler. The tokens can start in any column, and they do not have to be on the same line. The only requirement is that the opening token must precede the closing token. Figure 2-4 shows two examples of block comments.

```
/* This is a block comment that
   covers two lines.                */

/*
** It is a very common style to put the opening token
** on a line by itself, followed by the documentation
** and then the closing token on a separate line. Some
** programmers also like to put asterisks at the beginning
** of each line to clearly mark the comment.
*/
```

FIGURE 2-4 Examples of Block Comments

Line Comment

The second format, the **line comment**, uses two slashes (//) to identify a comment. This format does not require an end-of-comment token; the end of the line automatically ends the comment. Programmers generally use this format for short comments. The line-comment token can start anywhere on the line. Figure 2-5 contains two examples of line comments.

```
    // This is a whole line comment

    a = 5;            // This is a partial line comment
```

FIGURE 2-5 Examples of Line Comments

Although they can appear anywhere, comments cannot be nested. In other words, we cannot have comments inside comments. Once the compiler sees an opening block-comment token, it ignores everything it sees until it finds the closing token. Therefore, the opening token of the nested comment is not recognized, and the ending token that matches the first opening token is left standing on its own. This error is shown in Figure 2-6.

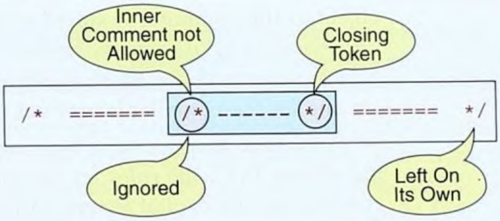


FIGURE 2-6 Nested Block Comments Are Invalid

The Greeting Program

Program 2-1 shows the greeting program just as we would write it. We have included some comments at the beginning that explain what the program is going to do. Each program we write begins with documentation explaining the purpose of the program. We have also shown comments to identify the declaration and statement sections of our program. The numbers on the left in Program 2-1 and the other programs in the text are for discussion reference. They are not part of the program.

PROGRAM 2-1 The Greeting Program

```
/* The greeting program. This program demonstrates
   some of the components of a simple C program.
      Written by:  your name here
      Date:        date program written
*/
#include <stdio.h>

int main (void)
{
// Local Declarations

// Statements

   printf("Hello World!\n");

   return 0;
} // main
```

2.3 Identifiers

One feature present in all computer languages is the **identifier**. Identifiers allow us to name data and other objects in the program. Each identified object in the computer is stored at a unique address. If we didn't have identifiers that we could use to symbolically represent data locations, we would have to know and use object's addresses. Instead, we simply give data identifiers and let the compiler keep track of where they are physically located.

Different programming languages use different syntactical rules to form identifiers. In C, the rules for identifiers are very simple. The only valid name symbols are the capital letters A through Z, the lowercase letters *a* through *z*, the digits 0 through 9, and the underscore. The first character of the identifier cannot be a digit.

Typically, application programs do not use the underscore for the first character either, because many of the identifiers in the C system libraries start with an underscore. In this way, we make sure that our names do not duplicate system names, which could become very confusing. The last rule is that the name we create cannot be **keywords**. Keywords, also known as **reserved words**, include syntactical words, such as *if* and *while*. For a list of the reserved words, see Appendix B.

Good identifier names are descriptive but short. To make them short, we often use abbreviations.[2] C allows names to be up to 63 characters long. If

[2] One way to abbreviate an identifier is to remove any vowels in the middle of the word. For example, *student* could be abbreviated *stdnt*.

the names are longer than 63 characters, then only the first 63 are used. Table 2-1 summarizes the rules for identifiers.

1. First character must be alphabetic character or underscore.
2. Must consist only of alphabetic characters, digits, or underscores.
3. First 63 characters of an identifier are significant.
4. Cannot duplicate a keyword.

TABLE 2-1 Rules for Identifiers

You might be curious as to why the underscore is included among the possible characters that can be used for an identifier. It is there so that we can separate different parts of an identifier. To make identifiers descriptive, we often combine two or more words. When the names contain multiple words, the underscore makes it easier to read the name.

An identifier must start with a letter or underscore: it may not have a space or a hyphen.

Another way to separate the words in a name is to capitalize the first letter in each word. The traditional method of separation in C uses the underscore. A growing group of programmers, however, prefer to capitalize the first letter of each word. Table 2-2 contains examples of valid and invalid names.

C is a case-sensitive language.

Two more comments about identifiers. Note that some of the identifiers in Table 2-2 are capitalized. Typically, capitalized names are reserved for preprocessor-defined names. The second comment is that C is case sensitive. This means that even though two identifiers are spelled the same, if the case of each corresponding letter doesn't match, C thinks of them as different names. Under this rule, `num`, `Num`, and `NUM` are three different identifiers.

Valid Names		Invalid Name	
a	// Valid but poor style	$sum	// $ is illegal
student_name		2names	// First char digit
_aSystemName		sum-salary	// Contains hyphen
_Bool	// Boolean System id	stdnt Nmbr	// Contains spaces
INT_MIN	// System Defined Value	int	// Keyword

TABLE 2-2 Examples of Valid and Invalid Names

2.4 Types

A **type** defines a set of values and a set of operations that can be applied on those values. For example, a light switch can be compared to a computer type. It has a set of two values, on and off. Only two operations can be applied to a light switch: turn on and turn off.

The C language has defined a set of types that can be divided into four general categories: void, integral, floating-point, and derived, as shown in Figure 2-7.

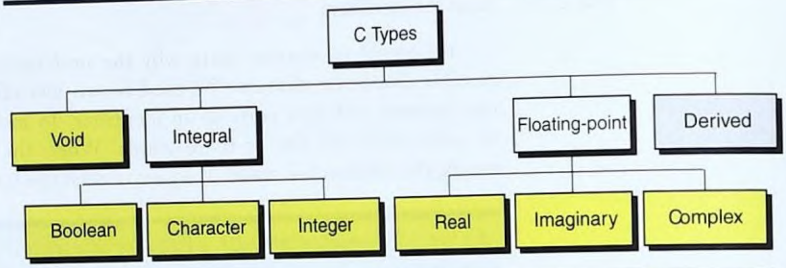


FIGURE 2-7 Data Types

In this chapter, we concentrate only on the first three types. The derived type will be discussed in future chapters.

Void Type

The void type, designated by the keyword *void*, has no values and no operations, Although having no values and operations might seem unusual, the *void* type is a very useful data type. For example, it is used to designate that a function has no parameters as we saw in the *main* function. It can also be used to define that a function has no return value as we see in Chapter 4. It can also be used to define a pointer to generic data as we will see in Chapter 9.

Integral Type

The C language has three **integral types**: Boolean, character, and integer. Integral types cannot contain a fraction part; they are whole numbers.

Boolean

With the release of C99, the C language incorporated a **Boolean** type. Named after the French mathematician/philosopher George Boole, a Boolean type can represent only two values: *true* or *false*. Prior to C99, C used integers to represent the Boolean values: a nonzero number (positive or negative) was used to represent true, and zero was used to represent false. For backward compatibility, integers can still be used to represent Boolean values; however, we recommend that new programs use the Boolean type. The Boolean type, which is referred to by the keyword *bool*, is stored in memory as 0 (*false*) or 1 (*true*).

Character

The third type is character. Although we think of characters as the letters of the alphabet, a computer has another definition. To a computer, a character is any value that can be represented in the computer's alphabet, or as it is better known, its **character set**. The C standard provides two character types: *char* and *wchar_t*.

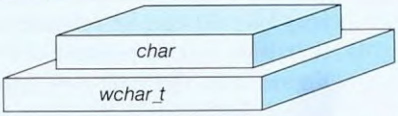


FIGURE 2-8 Character Types

Most computers use the American Standard Code for Information Interchange (**ASCII**—pronounced "ask-key") alphabet. You do not need to memorize this alphabet as you did when you learned your natural languages; however, you will learn many of the special values by using them. The ASCII code is included in Appendix A.

Most of the personal, mini-, and mainframe computers use 1 byte to store the *char* data types. A byte is 8 bits. With 8 bits, there are 256 different values in the *char* set. (Note in Appendix A that ASCII uses only half of these possible values.) Although the size of *char* is machine dependent and varies from computer to computer, normally it is 1 byte, or 8 bits.

If you examine the ASCII code carefully, you will notice that there is a pattern to its alphabet that corresponds to the English alphabet. The first 32 ASCII characters and the last ASCII character are control characters. They are used to control physical devices, such as monitors and printers, and in telecommunication systems. The rest are characters that we use to compose words and sentences.

All the lowercase letters are grouped together, as are all the uppercase letters and the digits. Many of the special characters, such as the shift characters on the top row of the keyboard, are grouped together, but some are found spread throughout the alphabet.

What makes the letter *a* different from the letter *x*? In English, it is the visual formation of the graphic associated with the letter. In the computer, it is the underlying value of the bit configuration for the letter. The letter *a* is binary `0110 0001`. The letter *x* is `0111 1000`. The decimal values of these two binary numbers are 97 and 120, respectively.

To support non-English languages and languages that don't use the Roman alphabet, the C99 standard created the wide character type (*wchar_t*). Without going in to all of the complexities, C supports two international standards, one for four-type characters and one for two-byte characters. Both of these standards support the traditional characters found in ASCII; that is, all extensions occur above the last ASCII character. The original ASCII characters are now known as the basic **Latin character set**. Generally speaking, the wide-character set is beyond the scope of an introductory programming text and is not covered in this text.

Integer

An integer type is a number without a fraction part. C supports four different sizes of the integer data type: *short int*, *int*, *long int*, and *long long int*. A *short int* can also be referred to as *short*, *long int* can be referred to as *long*, and *long long int* can be referred to as *long long*. C defines these data types so that they can be organized from the smallest to the largest, as shown in Figure 2-9. The type also defines the size of the field in which data can be stored. In C, this is true even though the size is machine dependent and varies from computer to computer.

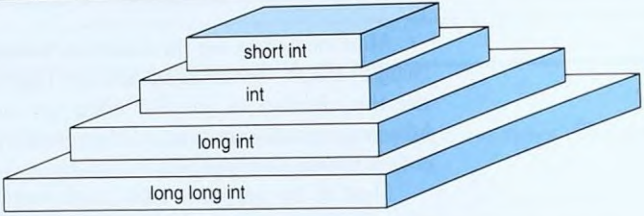


FIGURE 2-9 Integer Types

If we need to know the size of any data type, C provides an operator, *sizeof*, that will tell us the exact size in bytes. We will discuss this operator in detail in Chapter 3. Although the size is machine dependent, C requires that the following relationship always be true:

sizeof (short) ≤ sizeof (int) ≤ sizeof (long) ≤ sizeof (long long)

Each integer size can be a signed or an unsigned integer. If the integer is *signed*, then one bit must be used for a signed (0 is plus, 1 is minus). The *unsigned* integer can store a positive number that is twice as large as the signed integer of the same size.[3] Table 2-3 contains typical values for the integer types. Recognize, however, that the actual sizes are dependent on the physical hardware.

[3] For a complete discussion, see Appendix D, "Numbering Systems."

Type	Byte Size	Minimum Value	Maximum Value
short int	2	−32,768	32,767
int	4	−2,147,483,648	2,147,483,647
long int	4	−2,147,483,648	2,147,483,647
long long int	8	−9,223,372,036,854,775,807	9,223,372,036,854,775,806

TABLE 2-3 Typical Integer Sizes and Values for Signed Integers

To provide flexibility across different hardware platforms, C has a library, *limits.h*, that contains size information about integers. For example, the minimum integer value for the computer is defined as `INT_MIN`, and the maximum value is defined as `INT_MAX`. See Appendix E, "Integer and Float Libraries" for a complete list of these named values.

Floating-Point Types

The C standard recognizes three **floating-point types**: real, imaginary, and complex. Like the limits library for integer values, there is a standard library, *float.h*, for the floating-point values (see Appendix E, "Integer and Float Libraries"). Unlike the integral type, real type values are always signed.

Real

The **real type** holds values that consist of an integral and a fractional part, such as 43.32. The C language supports three different sizes of real types: *float*, *double*, and *long double*. As was the case for the integer type, real numbers are defined so that they can be organized from smallest to largest. The relationship among the real types is seen in Figure 2-10.

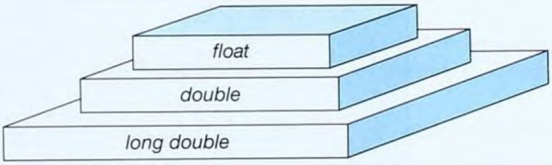


FIGURE 2-10 Floating-point Types

Regardless of machine size, C requires that the following relationship must be true:

sizeof (float) ≤ sizeof (double) ≤ sizeof (long double)

Imaginary Type

C defines an **imaginary type**. An imaginary number is used extensively in mathematics and engineering. An imaginary number is a real number multiplied by the square root of –1 (√–1). The imaginary type, like the real type, can be of three different sizes: *float imaginary*, *double imaginary*, and *long double imaginary*.

Most C implementations do not support the imaginary type yet and the functions to handle them are not part of the standard. We mention them here because the imaginary type is one of the components of the complex type.

Complex

C defines a **complex type**, which is implemented by most compilers. A complex number is a combination of a real and an imaginary number. The complex type, like the real type, can be of three different sizes: *float complex*, *double complex*, and *long long complex*. The size needs to be the same in both the real and the imaginary part. We provide two program examples that use complex numbers at the end of this chapter.

Type Summary

A summary of the four standard data types is shown in Table 2-4.

Category	Type	C Implementation
Void	Void	*void*
Integral	Boolean	*bool*
	Character	*char, wchar_t*
	Integer	*short int, int, long int, long long int*
Floating-Point	Real	*float, double, long double*
	Imaginary	*float imaginary, double imaginary, long double imaginary*
	Complex	*float complex, double complex, long double complex*

TABLE 2-4 Type Summary

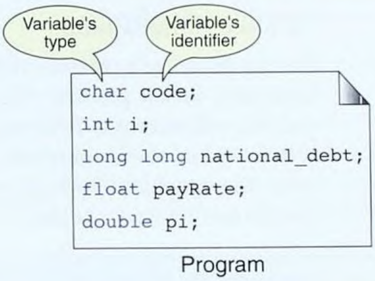
2.5 Variables

Variables are named memory locations that have a type, such as integer or character, which is inherited from their type. The type determines the values that a variable may contain the operations that may be used with its values.

Variable Declaration

Each variable in your program must be declared and defined. In C, a **declaration** is used to name an object, such as a variable. **Definitions** are used to create the object. With one exception, a variable is declared and defined at the same time. The exception, which we will see later, declares them first and then defines them at a later time. For variables, *definition* assumes that the declaration has been done or is being done at the same time. While this distinction is somewhat of an oversimplification, it works in most situations. We won't worry about the exception at this time.

When we create variables, the declaration gives them a symbolic name and the definition reserves memory for them. Once defined, variables are used to hold the data that are required by the program for its operation. Generally speaking, where the variable is located in memory is not a programmer's concern; it is a concern only of the compiler. From our perspective, all we are concerned with is being able to access the data through their symbolic names, their identifiers. The concept of variables in memory is illustrated in Figure 2-11.



```
char code;
int i;
long long national_debt;
float payRate;
double pi;
```

FIGURE 2-11 Variables

A variable's type can be any of the data types, such as *character, integer,* or *real*. The one exception to this rule is the type *void*; a variable cannot be type *void*.

To create a variable, we first specify the type, which automatically specifies it size (precision), and then its identifier, as shown below in the definition of a real variable named `price` of type float.

```
float price;
```

Table 2-5 shows some examples of variable declarations and definitions. As you study the variable identifiers, note the different styles used to make them readable. You should select a style and use it consistently. We prefer the use of an uppercase letter to identify the beginning of each word after the first one, although we do include examples using underscores.

```
bool   fact;
short  maxItems;              // Word separator: Capital
long   long national_debt;    // Word separator: underscore
float  payRate;               // Word separator: Capital
double tax;
float  complex voltage;
char   code, kind;            // Poor style—see text
int    a, b;                  // Poor style—see text
```

TABLE 2-5 Examples of Variable Declarations and Definitions

C allows multiple variables of the same type to be defined in one statement. The last two entries in Table 2-5 use this format. Even though many professional programmers use it, we consider it to be poor programming style. It is much easier to find and work with variables if they are defined on separate lines. This makes the compiler work a little harder, but the resulting code is no different. This is one situation in which ease of reading the program and programmer efficiency are more important than the convenience of coding multiple declarations on the same line.

Variable Initialization

We can initialize a variable at the same time that we declare it by including an initializer. When present, the **initializer** establishes the first value that the variable will contain. To initialize a variable when it is defined, the identifier is followed by the assignment operator[4] and then the initializer, which is the value the variable is to have when the function starts. This simple initialization format is shown below.

```
int count = 0;
```

Every time the function containing `count` is entered, `count` is set to zero. Now, what will be the result of the following initialization? Are both `count` and `sum` initialized or is only `sum` initialized?

```
int count, sum = 0;
```

The answer is that the initializer applies only to the variable defined immediately before it. Therefore, only `sum` is initialized! If you wanted both variables initialized, you would have to provide two initializers.

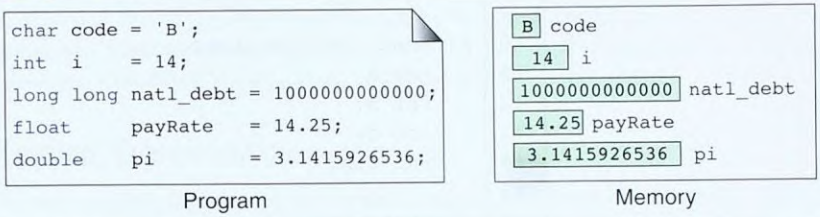
```
int count = 0, sum = 0;
```

[4] The assignment operator is the equal sign (=).

Again, to avoid confusion and error, we prefer using only one variable definition to a line. The preferred code in this case would be

```
int count =  0;
int sum   =  0;
```

Figure 2-12 repeats Figure 2-11, initializing the values in each of the variables.



```
char code = 'B';
int  i    = 14;
long long natl_debt = 1000000000000;
float     payRate   = 14.25;
double    pi        = 3.1415926536;
```

FIGURE 2-12 Variable Initialization

It is important to remember that with a few exceptions that we will see later, variables are not initialized automatically. When variables are defined, they usually contain garbage (meaningless values left over from a previous use), so we need to initialize them or store data in them (using run-time statements) before accessing their values. Many compilers display a warning message when a variable is accessed before it is initialized.

When a variable is defined, it is not initialized. We must initialize any variable requiring prescribed data when the function starts.

One final point about initializing variables when they are defined: Although the practice is convenient and saves you a line of code, it also can lead to errors. It is better, therefore, to initialize the variable with an assignment statement at the proper place in the body of the code. This may take another statement, but the efficiency of the resulting program is exactly the same, and you will make fewer errors in your code.

EXAMPLE 2-1 Print Sum

At this point you might like to see what a more complex program looks like. As you read Program 2-2, note the blank lines to separate different groups of code. This is a good technique for making programs more readable. You should use blank lines in your programs the same way you use them to separate the paragraphs in a report.

PROGRAM 2-2 Print Sum of Three Numbers

```c
/* This program calculates and prints the sum of
   three numbers input by the user at the keyboard.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int a;
   int b;
   int c;
   int sum;

// Statements
   printf("\nWelcome. This program adds\n");
   printf("three numbers. Enter three numbers\n");
   printf("in the form: nnn nnn nnn <return>\n");
   scanf("%d %d %d", &a, &b, &c);

   // Numbers are now in a, b, and c. Add them.
   sum = a + b + c;

   printf("The total is: %d\n\n", sum);

   printf("Thank you. Have a good day.\n");
   return 0;
} //  main
```

```
Results:
Welcome. This program adds
three numbers. Enter three numbers
in the form: nnn nnn nnn <return>
11 22 33

The total is: 66

Thank you. Have a good day.
```

Program 2-2 Analysis Study the style of this program carefully. First, note how we start with a welcome message that tells the user exactly what needs to be entered. Similarly, at the end of the program, we print an ending message. It is a good style to print a start and end message.

This program contains three different processes. First it reads three numbers. The code to read the numbers includes the printed instructions and a read (*scanf*) statement. The second process adds the three numbers. While this process consists of only a comment and one statement, we separate it from the read process. This makes it easier for the reader to follow the program. Finally, we print the result. Again, the print process is separated from the calculate process by a blank line.

2.6 Constants

Constants are data values that cannot be changed during the execution of a program. Like variables, constants have a type. In this section, we discuss Boolean, character, integer, real, complex, and string constants.

Constant Representation

In this section, we show how we use symbols to represent constants. In the next section, we show how we can code these constants in our program.

Boolean Constants

A Boolean data type can take only two values. Therefore, we expect that we have only two symbols to represent a Boolean type. The values are *true* and *false*. As we mentioned before, a Boolean value can have only one of the two values: 0 (*false*) and 1 (*true*). We use the constant *true* or *false* in our program. To do so, however, requires that we include the Boolean library, *stdbool.h*.

Character Constants

Character constants are enclosed between two single quotes (apostrophes). In addition to the character, we can also use a backslash (\) before the character. The backslash is known as the **escape character**. It is used when the character we need to represent does not have any graphic associated with it—that is, when it cannot be printed or when it cannot be entered from the keyboard. The escape character says that what follows is not the normal character but something else. For example, `'\n'` represents the newline character (line feed). So, even though there may be multiple symbols in the character constant, they always represent only one character.

A character constant is enclosed in single quotes.

Wide-character constants are coded by prefixing the constant with an `L`, as shown in the following example.

```
L'x'
```

The character in the character constant comes from the character set supplied by the hardware manufacturer. Most computers use the ASCII character set, or as it is sometimes called, the ASCII alphabet. The ASCII character set is shown in Appendix A.

C has named the critical character values so that we can refer to them symbolically. Note that these control characters use the escape character followed by a symbolic character. They are shown in Table 2-6.

ASCII Character	Symbolic Name
null character	`'\0'`
alert (bell)	`'\a'`
backspace	`'\b'`
horizontal tab	`'\t'`
newline	`'\n'`
vertical tab	`'\v'`
form feed	`'\f'`
carriage return	`'\r'`
single quote	`'\''`
double quote	`'\"'`
backslash	`'\\'`

TABLE 2-6 Symbolic Names for Control Characters

Integer Constants

Although integers are always stored in their binary form, they are simply coded as we would use them in everyday life. Thus, the value 15 is simply coded as `15`.

If we code the number as a series of digits, its type is signed integer, or signed long integer if the number is large. We can override this default by specifying unsigned (`u` or `U`), and *long* (`l` or `L`) or *long long* (`ll` or `LL`), after the number. The codes may be combined and may be coded in any order. Note that there is no way to specify a *short int* constant. When we omit the suffix on a literal, it defaults to *int*. While both upper- and lowercase codes are allowed, we recommend that you always use uppercase to avoid confusion (especially with the lowercase letter *l*, which often looks like the number 1). Table 2-7 shows several examples of integer constants. The default types are typical for a personal computer.

Representation	Value	Type
+123	123	int
–378	–378	int
–32271L	–32,271	long int
76542LU	76,542	unsigned long int
12789845LL	12,789,845	long long int

TABLE 2-7 Examples of Integer Constants

Real Constants

The default form for real constants is *double*. If we want the resulting data type to be *float* or *long double*, we must use a code to specify the desired data type. As you might anticipate, `f` and `F` are used for *float* and `l` and `L` are used for *long double*. Again, do not use the lowercase *l* for *long double*; it is too easily confused with the number 1.

Table 2-8 shows several examples of real constants.

Representation	Value	Type
0.	0.0	double
.0	0.0	double
2.0	2.0	double
3.1416	3.1416	double
–2.0f	–2.0	float
3.1415926536L	3.1415926536	long double

TABLE 2-8 Examples of Real Constants

Complex Constants

Although we do not discuss the imaginary constants, we need to talk about complex constants that are widely used in engineering.

Complex constants are coded as two parts, the real part and the imaginary part, separated by a plus sign. The real part is coded using the real format rules. The imaginary part is coded as a real number times (`*`) the imaginary constant (`_Complex_I`). If the complex library (*complex.h*) is included, the imaginary constant can be abbreviated as `I`. Examples are shown in Table 2-9.

Representation	Value	Type
12.3 + 14.4 * I	12.3 + 14.4 * $(-1)^{1/2}$	double complex
14F + 16F * I	14 + 16 * $(-1)^{1/2}$	float complex
1.4736L+ 4.56756L * I	1.4736 + 4.56756 * $(-1)^{1/2}$	long double complex

TABLE 2-9 Examples of Complex Constants

The default form for complex constants is *double*. If we want the resulting data type to be *float* or *long double*, we must use a code to specify the desired data type. As you might anticipate, `f` and `F` are used for *float* and `l` and `L` are used for *long double*. Again, do not use the lowercase *l* for *long double*; it is too easily confused with the number 1.

The two components of a complex constant must be of the same precision, that is, if the real part is type double, then the imaginary time must also be type double.

Table 2-9 shows several examples of complex constants. Note that we use the abbreviated form for the imaginary part.

String Constants

A string constant is a sequence of zero or more characters enclosed in double quotes. You used a string in your first program without even knowing that it was a string! Look at Program 2-1 to see if you can identify the string.

Listed in Figure 2-13 are several strings, including the one from Program 2-1. The first example, an empty string, is simply two double quotes in succession. The second example, a string containing only the letter `h`, differs from a character constant in that it is enclosed in double quotes. When we study strings, we will see that there is also a big difference in how `h` is stored in memory as a character and as a string. The last example is a string that uses wide characters.

```
""
"h"                                         // A null string
"Hello World\n"
"HOW ARE YOU"
"Good Morning!"
L"This string contains wide characters."
```

FIGURE 2-13 Some Strings

It is important to understand the difference between the null character (see Table 2-6) and an empty string. The null character represents no value. As a character, it is 8 zero bits. An empty string, on the other hand, is a string containing nothing. Figure 2-14 shows the difference between these two constant types.

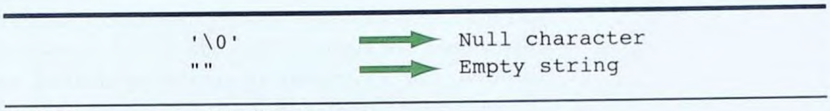


FIGURE 2-14 Null Characters and Null Strings

At this point, this is all you need to know about strings. We talk more about them and how they are stored in the computer when we study strings in Chapter 11.

Use single quotes for character constants. Use double quotes for string constants.

Coding Constants

In this section we discuss three different ways we code constants in our programs: literal constants, defined constants, and memory constants.

Literal Constants

A **literal** is an unnamed constant used to specify data. If we know that the data cannot be changed, then we can simply code the data value itself in a statement.

Literals are coded as part of a statement using the constant formats described in the previous section. For example, the literal 5 is used in the following statement.

```
a = b + 5;
```

Defined Constants

Another way to designate a constant is to use the preprocessor command define. Like all preprocessor commands, it is prefaced with the pound sign (#). The define commands are usually placed at the beginning of the program, although they are legal anywhere. Placing them at the beginning of the program makes them easy to find and change. A typical define command might be

```
#define SALES_TAX_RATE .0825
```

In the preceding example, for instance, the sales tax rate changes more often than we would like. By placing it and other similar constants at the beginning of the program, we can find and change them easily.

As the preprocessor reformats the program for the language translator, it replaces each defined name, `SALES_TAX_RATE` in the previous example with its value (`.0825`) wherever it is found in the source program. This action is just like the search-and-replace command found in a text editor. The preprocessor does not evaluate the code in any way—it just blindly makes the substitution. For a complete discussion of defined constants, see Appendix G, "Preprocessor Commands."

Memory Constants

The third way to use a constant is with memory constants. Memory constants use a C **type qualifier**, *const*, to indicate that the data cannot be changed. Its format is:

```
const type identifier = value;
```

We have seen how to define a variable, which does nothing more than give a type and size to a named object in memory. Now let us assume that we want to fix the contents of this memory location so that they cannot be changed. This is the same concept as a literal, only now we give it a name. The following code creates a memory constant, `cPi`. To help us remember that it is a constant, we preface the identifier name with *c*.

```
const float cPi = 3.14159;
```

Three points merit discussion: (1) The type qualifier comes first. (2) Then there must be an initializer. If we didn't have an initializer, then our named constant would be whatever happened to be in memory at `cPi`'s location when our program started. (3) Finally, since we have said that `cPi` is a constant, we cannot change it.

Program 2-3 demonstrates the three different ways to code `pi` as a constant.

PROGRAM 2-3 Memory Constants

```
/* This program demonstrates three ways to use
   constants.
      Written by:
      Date:
*/
#include <stdio.h>
#define PI 3.1415926536

```

continued

PROGRAM 2-3 Memory Constants *(continued)*

```
int main (void)
{
// Local Declarations
   const double cPi = PI;

// Statements
   printf("Defined constant PI: %f\n", PI);
   printf("Memory constant cPi: %f\n", cPI);
   printf("Literal constant:    %f\n", 3.1415926536);
   return 0;
}  // main
```

```
Results:
Defined constant PI:  3.141593
Memory constant cPi:  3.141593
Literal constant:     3.141593
```

2.7 Input/Output

Although our programs have implicitly shown how to print messages, we have not formally discussed how we use C facilities to input and output data. We devote two chapters, Chapters 7 and 13, to fully explain the C input/output facilities and how to use them. In this section, we describe simple input and output formatting.

Streams

In C, data is input to and output from a stream. A **stream** is a source of or destination for data. It is associated with a physical device, such as a terminal, or with a file stored in auxiliary memory.

C uses two forms of streams: text and binary. A **text stream** consists of a sequence of characters divided into lines with each line terminated by a newline (\n). A **binary stream** consists of a sequence of data values such as integer, real, or complex using their memory representation. In this chapter, we briefly discuss only text streams. A more detailed discussion of text streams is found in Chapter 7, "Text Input/Output," and a detailed discussion of binary streams is found in Chapter 13, "Binary Input/Output."

A terminal can be associated only with a text stream because a keyboard can only send a stream of characters into a program and a monitor can only display a sequence of characters. A file, on the other hand can be associated with a text or binary stream. We can store data in a file and later retrieve them as a sequence of characters (text stream) or as a sequence of data values (binary streams).

In this chapter, we assume that the source of data is the keyboard and the destination of data is the monitor. In other words, the terminal devices we use produce or consume text streams. In C, the keyboard is known as **standard input** and the monitor is known as **standard output**.

A terminal keyboard and monitor can be associated only with a text stream. A keyboard is a source for a text stream; a monitor is a destination for a text stream.

Figure 2-15 shows the concept of streams and the two physical devices associated with input and output text streams.

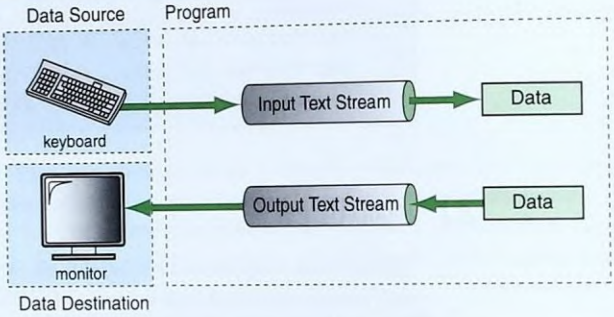


FIGURE 2-15 Stream Physical Devices

Formatting Input/Output

The previous section discussed the terminal as a text stream source and destination. We can only receive text streams from a terminal (keyboard) and send text streams to a terminal (monitor). However, these text streams often represent different data types, such as integer, real, and Boolean. The C language provides two formatting functions: *printf* for output formatting and *scanf* for input formatting. The *printf* function converts data stored in the program into a text stream for output to the monitor; the *scanf* function converts the text stream coming from the keyboard to data values and stores them in program variables. In other words, the *printf* and *scanf* functions are data to text stream and text stream to data converters.

Output Formatting: *printf*

The output formatting function is *printf*. The *printf* function takes a set of data values, converts them to a text stream using formatting instructions contained in a format control string, and sends the resulting text stream to the standard output (monitor). For example, an integer 234 stored in the program is converted to a text stream of three numeric ASCII characters ('2', '3', and '4') and then is sent to the monitor. What we see on the monitor is these three characters, not the integer 234. However, we interpret the three characters together as an integer value. Figure 2-16 shows the concept.

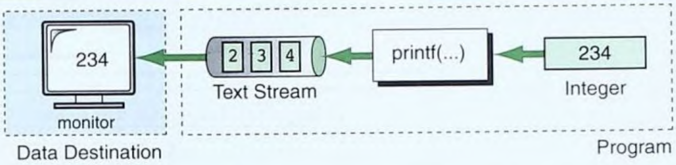


FIGURE 2-16 Output Formatting Concept

Basic Concept

The *printf* function uses an interesting design to convert data into text streams. We describe how the text stream should be formatted using a **format control string** containing zero or more **conversion specifications**. In addition to the conversion specifications, the control string may contain textual data and control characters to be displayed.

Each data value to be formatted into the text stream is described as a separate conversion specification in the control string. The specifications describe the data values' type, size, and specific format information, such as how wide the display width should be. The location of the conversion specification within the format control string determines its position within the text stream.

The control string and data values are passed to the print function (*printf*) as parameters, the control string as the first parameter and one parameter for each value to be printed. In other words, we supply the following information to the print function:

1. The format control string including any textual data to be inserted into the text stream.
2. A set of zero or more data values to be formatted.

Figure 2-17 is a conceptional representation of the format control string and two conversion specifications.

Figure 2-17(a) shows the format string and the data values as parameters for the print function. Within the control string we have specified quantity (`Qty:`) and total (`Tot:`) as textual data and two conversion specifications (`%d` and `%f`). The first specification requires an integer type value; the second requires a real type value. We discuss the conversion specifications in detail in the following section.

Figure 2-17(b) shows the formatting operation and the resulting text stream. The first data value is a literal integer; the second data value is the contents of a variable named `tot`. This part of Figure 2-17 shows how the *printf* function expands the control stream and inserts the data values and text characters.

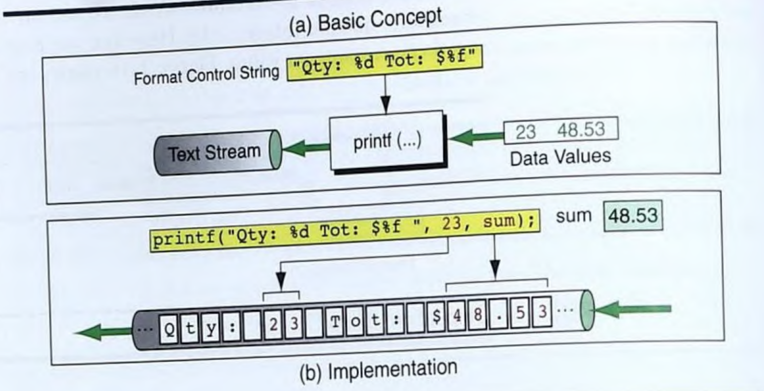


FIGURE 2-17 Output Stream Formatting Example

Format Control String Text

The control string may also contain text to be printed, such as instructions to the user, captions or other identifiers, and other text intended to make the output more readable. In fact, as we have already seen, the format string may contain nothing but text, in which case the text will be printed exactly as shown. We used this concept when we wrote the greeting program. In addition, we can also print control characters, such as tabs (`\t`), newlines (`\n`), and alerts (`\a`), by including them in the format string. Tabs are used to format the output into columns. Newlines terminate the current line and continue formatting on the next line. Alerts sound an audio signal to alert, usually to alert the user to a condition that needs attention. These control characters are seen in Table 2-6.

Conversion Specification

To insert data into the stream, we use a conversion specification that contains a start token (`%`), a **conversion code**, and up to four optional modifiers as shown in Figure 2-18. Only the field-specification token (`%`) and the conversion code are required.

%	Flag	Minimum Width	Precision	Size	Code

FIGURE 2-18 Conversion Specification

Approximately 30 different conversion codes are used to describe data types. For now, however, we are concerned with only three: character (`c`), integer (d), and floating point (f). These codes with some examples are shown in Table 2-10.

Type	Size[a]	Code	Example
char	None	c	%c
short int	h	d	%hd
int	None	d	%d
long int	None	d	%ld
long long int	ll	d	%lld
float	None	f	%f
double	None	f	%f
long double	L	f	%Lf

[a] Size is discussed in the next section.

TABLE 2-10 Format Codes for Output

The **size modifier** is used to modify the type specified by the conversion code. There are four different sizes: h, l (el), ll (el el), and L. The h is used with the integer codes to indicate a short integer value;[5] the l is used to indicate a long integer value; the ll is used to indicate a long long integer value; and the L is used with floating-point numbers to indicate a long double value.

A **width modifier** may be used to specify the minimum number of positions in the output. (If the data require using more space than we allow, then *printf* overrides the width.) It is very useful to align output in columns, such as when we need to print a column of numbers. If we don't use a width modifier, each output value will take just enough room for the data.

If a floating-point number is being printed, then we may specify the number of decimal places to be printed with the **precision modifier**. The precision modifier has the format

```
.m
```

where m is the number of decimal digits. If no precision is specified, *printf* prints six decimal positions. These six decimal positions are often more than is necessary.

When both width and precision are used, the width must be large enough to contain the integral value of the number, the decimal point, and the number of digits in the decimal position. Thus, a conversion specification

[5] The h code is a carry-over from assembler language where it meant half word.

of `%7.2f` is designed to print a maximum value of 9999.99. Some examples of width specifications and precision are shown below.

```
%2hd          // short integer–2 print positions
%4d           // integer–4 print positions
%8ld          // long int–8 (not 81) positions
%7.2f         // float–7 print positions: nnnn.dd
%10.3Lf       // long double–10 positions: nnnnnn.ddd
```

The **flag modifier** is used for four print modifications: justification, padding, sign, and numeric conversion variants. The first three are discussed here; the conversion variants are discussed in Chapter 7.

Justification controls the placement of a value when it is shorter than the specified width. Justification can be left or right. If there is no flag and the defined width is larger than required, the value is right justified. The default is right justification. To left justify a value, the flag is set to minus (-).

Padding defines the character that fills the unused space when the value is smaller than the print width. It can be a space, the default, or zero. If there is no flag defined for padding, the unused width is filled with spaces; if the flag is `0`, the unused width is filled with zeroes. Note that the zero flag is ignored if it is used with left justification because adding zeros after a number changes its value.

The **sign flag** defines the use or absence of a sign in a numeric value. We can specify one of three formats: default formatting, print signed values, or prefix positive values with a leading space. Default formatting inserts a sign only when the value is negative. Positive values are formatted without a sign. When the flag is set to a plus (+), signs are printed for both positive and negative values. If the flag is a space, then positive numbers are printed with a leading space and negative numbers with a minus sign.

Table 2-11 documents three of the more common flag options.

Flag Type	Flag Code	Formatting
Justification	None	right justified
	–	left justified
Padding	None	space padding
	0	zero padding
Sign	None	positive value: no sign negative value: –
	+	positive value: + negative value: –
	None	positive value: space negative value: –

TABLE 2-11 Flag Formatting Options

Output Examples

This section contains several output examples. We show the *printf* statement, followed by what would be printed. Cover up the solution and try to predict the results.

1. `printf("%d%c%f", 23, 'z', 4.1);`

   ```
   23z4.100000
   ```

   Note that because there are no spaces between the conversion specifications, the data are formatted without spaces between the values.

2. `printf("%d %c %f", 23, 'z', 4.1);`

   ```
   23 z 4.100000
   ```

   This is a repeat of Output Example 1 with spaces between the conversion specifications.

3.
   ```
   int   num1  = 23;
   char  zee   = 'z';
   float num2  = 4.1;
   printf("%d %c %f", num1, zee, num2);
   ```

   ```
   23 z 4.100000
   ```

   Again, the same example, this time using variables.

4.
   ```
   printf("%d\t%c\t%5.1f\n", 23, 'Z', 14.2);
   printf("%d\t%c\t%5.1f\n", 107, 'A', 53.6);
   printf("%d\t%c\t%5.1f\n", 1754, 'F', 122.0);
   printf("%d\t%c\t%5.1f\n", 3, 'P', 0.1);
   ```

   ```
   23      Z     14.2
   107     A     53.6
   1754    F    122.0
   3       P      0.1
   ```

   In addition to the conversion specifications, note the tab character (`\t`) between the first and second, and second and third conversion specifications. Since the data are to be printed in separate lines, each format string ends with a newline (`\n`).

5. `printf("The number%dis my favorite number.", 23);`

   ```
   The number23is my favorite number.
   ```

   Since there are no spaces before and after the format code (`%d`), the number 23 is run together with the text before and after.

6. `printf("The number is %6d", 23);`

```
The number is     23
^^^^^^^^^^^^^^^^^^^^
```

If you count the spaces carefully, you will note that five spaces follow the word `is`. The first space comes from the space after `is` and before the `%` in the format string. The other four come from the width in the conversion specification.

7. `printf("The tax is %6.2f this year.", 233.12);`

```
The tax is 233.12 this year.
```

In this example, the width is six and the precision two. Since the number of digits printed totals five (three for the integral portion and two for the decimal portion), and the decimal point takes one print position, the full width is filled with data. The only spaces are the spaces before and after the conversion code in the format string.

8. `printf("The tax is %8.2f this year.", 233.12);`

```
The tax is   233.12 this year.
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

9. `printf("The tax is %08.2f this year.", 233.12);`

```
The tax is 00233.12 this year.
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

This example uses the zero flag to print leading zeros. Note that the width is eight positions. Three of these positions are taken up by the precision of two digits and the decimal point. This leaves five positions for the integral portion of the number. Since there are only three digits (233), *printf* inserts two leading zeros.

10. `printf("\"%8c   %d\"", 'h', 23);`

```
"       h   23"
^^^^^^^^^^^^^^^
```

In this example, we want to print the data within quotes. Since quotes are used to identify the format string, we can't use them as print characters. To print them, therefore, we must use the escape character with the quote (`\"`), which tells *printf* that what follows is not the end of the string but a character to be printed, in this case, a quote mark.

11.
```
printf ("This line disappears.\r...A new line\n");
printf ("This is the bell character \a\n");
printf ("A null character\0kills the rest of the line\n");
printf ("\nThis is \'it\' in single quotes\n");
printf ("This is \"it\" in double quotes\n");
printf ("This is \\ the escape character itself\n");
```

```
...A new line
This is the bell character
A null character
This is 'it' in single quotes
This is "it" in double quotes
This is \ the escape character itself
```

These examples use some of the control character names found in Table 2-6. Two of them give unexpected results. In Output Example 11, the return character (\r) repositions the output at the beginning of the current line without advancing the line. Therefore, all data that were placed in the output stream are erased.

The null character effectively kills the rest of the line. Had we not put a newline character (\n) at the beginning of the next line, it would have started immediately after `character`.

12. New example with multiple flags.

```
printf("|%-+8.2f| |%0+8.2f| |%-0+8.2f|", 1.2, 2.3, 3.4);
```

```
|+1.20   | |+0002.30| |+3.40   |
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

This example uses multiple flags. So that we can see the justification, each value is enclosed in vertical bars. The first value is printed left justified with the positive flag set. The second example uses zero fill with a space for the sign. Note that there is a leading space in the output. This represents the plus value. It is then followed by the leading zeros. The last example demonstrates that the zero fill is ignored when a numeric value is printed with left justification.

Common Output Errors

Each of the following examples has at least one error. Try to find each one before you look at the solutions. Your results may vary depending on your compiler and hardware.

1. `printf ("%d %d %d\n", 44, 55);`

```
44 55 0
```

This example has three conversion specifications but only two values.

2. `printf ("%d %d\n", 44, 55, 66);`

```
44 55
```

This example has two conversion specifications with three values. In this case, *printf* ignores the third value.

3.
```
float x = 123.45;
printf("The data are: %d\n", x);
```

```
The data are: 1079958732
```

This is a very common error in which the format specification (integer) does not match the data type (real).

Input Formatting: *scanf*

The standard input formatting function in C is *scanf* (scan formatting). This function takes a text stream from the keyboard, extracts and formats data from the stream according to a format control string, and then stores the data in specified program variables. For example, the stream of 5 characters '2', '3', '4', '.', and '2' are extracted as the real 234.2. Figure 2-19 shows the concept.

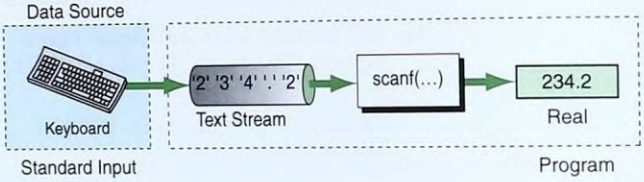


FIGURE 2-19 Formatting Text from an Input Stream

The *scanf* function is the reverse of the *printf* function.

1. A format control string describes the data to be extracted from the stream and reformatted.
2. Rather than data values as in the *printf* function, *scanf* requires the *variable* addresses were each piece of data are to be stored. Unlike the *printf* function, the destination of the data items cannot be literal values, they must store in the variables.
3. With the exception of the character specification, leading whitespaces are discarded.
4. Any non-conversion specification characters in the format string must be exactly matched by the next characters in the input stream.

   We must be careful about extra characters in the control stream. Extra characters in the control string can be divided into two categories: non-whitespace and whitespace.

a. Non-whitespace characters in the control string must exactly match characters entered by the user and are discarded by the *scanf* after they are read. If they don't match, then *scanf* goes into an error state and the program must be manually terminated.

   We recommend that you don't use non-whitespace characters in the format string, at least until you learn how to recover from errors in Chapter 13. However, there are some uses for it. For example, if the users want to enter dates with slashes, such as 5/10/06, the slashes must either be read and discarded using the character format specification (see the discussion of the assignment suppression flag in the later section, "Conversion Specification") or coded as non-whitespace in the format specification. We prefer the option to read and discard them.

b. Whitespace characters in the format string are matched by zero or more whitespace characters in the input stream and discarded. There are two exceptions to this rule: the character conversion code and the scan set (see Chapter 11) do not discard whitespace. It is easy, however, to manually discard whitespace characters when we want to read a character. We simply code a space before the conversion specification or as a part of it as shown below. Either one works.

```
" %c" or "% c"
```

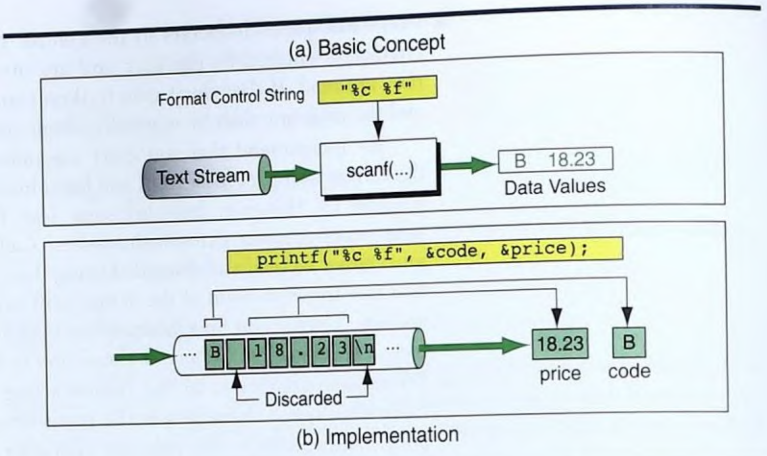
Remember that whenever we read data from the keyboard, there is a return character from a previous read. If we don't flush the whitespace characters when we read a character, therefore, we will get the whitespace from the previous read. To read a character, we should always code at least one whitespace character in the conversion specification; otherwise the whitespace remaining in the input stream is read as the input character. For example, to read three characters, we should code the following format string. Note the spaces before each conversion specification.

```
scanf(" %c %c %d", &c1, &c2, &c3);
```

Figure 2-20 demonstrates the input format string concept with a control string having two fields (%d and %f). The first one defines that a character will be inserted here; the second defines that a real will be inserted there. We will discuss these place holders, or format specifiers, later in the chapter.

Format Control String

Like the control string for *printf*, the control string for *scanf* is enclosed in a set of quotation marks and contains one or more conversion specifications that describe the data types and indicate any special formatting rules and/or characters.



(b) Implementation

FIGURE 2-20 Input Stream Formatting Example

Conversion Specification

To format data from the input stream, we use a conversion specification that contains a start token (%), a **conversion code**, and up to three optional modifiers as shown in Figure 2-21. Only the field-specification token (%) and the conversion code are required.

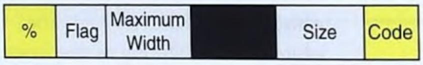


FIGURE 2-21 Conversion Specification

There are only three differences between the conversion codes for input formatting and output formatting. First, there is no precision in an input conversion specification. It is an error to include a precision; if *scanf* finds a precision it stops processing and the input stream is in the error state.

There is only one flag for input formatting, the assignment suppression flag (*). More commonly associated with text files (see Chapter 7), the assignment suppression flag tells *scanf* that the next input field is to be read but not stored. It is discarded. The following *scanf* statement reads an integer, a character, and a floating-point number from the input stream. The character is read and discarded. The other fields are read, formatted, and stored. Note that there is no matching address parameter for the data to be discarded.

```
scanf ("%d %*c %f", &x, &y);
```

The third difference is the width specification; with input formatting it is a *maximum*, not a minimum, width. The width modifier specifies the maximum number of characters that are to be read for one format code. When a width specification is included, therefore, *scanf* reads until the maximum number of characters have been processed or until *scanf* finds a whitespace character. If *scanf* finds a whitespace character before the maximum is reached, it stops.

Input Parameters

For every conversion specification there must be a matching variable in the address list. The address list contains the address of the matching variable. How do we specify an address? It's quite simple: Addresses are indicated by prefixing the variable name with an ampersand (`&`). In C, the ampersand is known as the **address operator**. Using the address operator, if the variable name is `price`, then the address is `&price`. Forgetting the ampersand is one of the most common errors for beginning C programmers, so you will have to concentrate on it when you use the *scanf* function.

scanf requires variable addresses in the address list.

Remember that the first conversion specification matches the first variable address, the second conversion specification matches the second variable address, and so on. This correspondence is very important. It is also very important that the variable's type match the conversion specification type. The C compiler does not verify that they match. If they don't, the input data will not be properly formatted when they are stored in the variable.

End of File and Errors

In addition to whitespace and width specifications, two other events stop the *scanf* function. If the user signals that there is no more input by keying **end of file** (EOF), then *scanf* terminates the input process. While there is no EOF on the keyboard, it can be simulated in most systems. For example, Windows uses the `<ctrl + z>` key combination to signal EOF. Unix and Apple Macintosh use `<ctrl + d>` for EOF. The C user's manual for your system should specify the key sequence for EOF.

Second, if *scanf* encounters an invalid character when it is trying to convert the input to the stored data type, it stops. The most common error is finding a nonnumeric character when it is trying to read a number. The valid characters are leading plus or minus, digits, and one decimal point. Any other combination, including any alphabetic characters, will cause an error. Although it is possible to detect this error and ask the user to re-input the data, we will not be able to cover the conventions for this logic until Chapter 7. Until then, be very careful when you enter data into your program.

Input Formatting Summary

Table 2-12 summarizes the rules for using *scanf*.

1. The conversion operation processes until:
    a. End of file is reached.
    b. The maximum number of characters has been processed.
    c. A whitespace character is found after a digit in a numeric specification.
    d. An error is detected.
2. There must be a conversion specification for each variable to be read.
3. There must be a variable address of the proper type for each conversion specification.
4. Any character in the format string other than whitespace or a conversion specification must be exactly matched by the user during input. If the input stream does not match the character specified, an error is signaled and *scanf* stops.
5. It is a fatal error to end the format string with a whitespace character. Your program will not run correctly if you do.

TABLE 2-12 *scanf* Rules

Input Examples

This section contains several examples. We list the data that will be input first. This allows you to cover up the function and try to formulate your own *scanf* statement.

1. `214 156 14Z`

```
scanf("%d%d%d%c", &a, &b, &c, &d);
```

Note that if there were a space between the `14` and the `Z`, it would create an error because `%c` does not skip whitespace! To prevent this problem, put a space before the `%c` code as shown below. This will cause it to skip leading whitespace.

```
scanf("%d%d%d %c", &a, &b, &c, &d);
```

2. `2314  15  2.14`

```
scanf("%d %d %f",  &a, &b, &c);
```

Note the whitespace between the conversion specifications. These spaces are not necessary with numeric input, but it is a good idea to include them.

3. `14/26  25/66`

```
scanf("%2d/%2d %2d/%2d",
      &num1, &den1, &num2, &den2);
```

Note the slashes (/) in the format string. Since they are not a part of the conversion specification, the user must enter them exactly as shown or *scanf* will stop reading.

4. `11-25-56`

```
scanf ("%d-%d-%d", &a, &b, &c);
```

Again, we see some required user input, this time dashes between the month, day, and year. While this is a common date format, it can cause problems. A better solution would be to prompt the user separately for the month, the day, and the year.

Common Input Errors

Each of the following examples has at least one error. Try to find it before you look at the solution. Your results may vary depending on your compiler and hardware.

1.
```
int a = 0;
scanf ("%d",   a);
printf("%d\n", a);
```

```
234                                  (Input)
0                                    (Output)
```

This example has no address token on the variable (`&a`). If the program runs at all, the data are read into an unidentified area in memory. What is printed is the original contents of the variable, in this case `0`.

2.
```
float a = 2.1;
scanf  ("%5.2f", &a);
printf ("%5.2f",  a);
```

```
74.35                                (Input)
2.10                                 (Output)
```

This example has no precision in the input conversion specification. When *scanf* finds a precision, it stops processing and returns to the function that called it. The input variable is unchanged.

3.
```
int a;
int b;
scanf  ("%d%d%d", &a , &b);
printf ("%d %d\n", a ,  b);
```

```
5 10                              (input)
5 10                              (output)
```

This example has three conversion specifications but only two addresses. Therefore, *scanf* reads the first two values and quits because no third address if found.

4.
```
int a = 1;
int b = 2;
int c = 3;
scanf  ("%d%d",   &a , &b, &c);
printf ("%d %d\n", a ,  b,  c);
```

```
5 10 15                           (input)
5 10 3                            (output)
```

This example has only two conversion specifications, but it has three addresses. Therefore, *scanf* reads the first two values and ignores the third address. The value 15 is still in the input stream waiting to be read.

2.8 Programming Examples

In this section, we show some programming example to emphasize the ideas and concepts we have discussed about input/output.

EXAMPLE 2-2 Print "Nothing!"

Program 2-4 is a very simple program that prints "`Nothing!`"

PROGRAM 2-4 A Program That Prints "Nothing!"

```
/* Prints the message "Nothing!".
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Statements
   printf("This program prints\n\n\t\"Nothing!\"");
   return 0;
} // main
```

continued

PROGRAM 2-4 A Program That Prints "Nothing!" *(continued)*

```
Results:
This program prints

    "Nothing!"
```

EXAMPLE 2-3 Print Boolean Constants

Program 2-5 demonstrates printing Boolean values. As the program shows, however, while a Boolean literal contains either *true* or *false*, when it is printed, it is printed as 0 or 1. This is because there is no conversion code for Boolean. To print it, we must use the integer type, which prints its stored value, 0 or 1.

PROGRAM 2-5 Demonstrate Printing Boolean Constants

```c
/* Demonstrate printing Boolean constants.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdbool.h>

int main (void)
{
// Local Declarations
   bool x = true;
   bool y = false;

// Statements
   printf ("The Boolean values are: %d %d\n", x, y);
   return 0;
}  // main
```

```
Results:
The Boolean values are: 1 0
```

EXAMPLE 2-4 Print Character Values

Program 2-6 demonstrates that all characters are stored in the computer as integers. We define some character variables and initialize them with values, and then we print them as integers. As you study the output, note that the ASCII values of the characters are printed. The program also shows the value of some nonprintable characters. All values can be verified by referring to Appendix A.

PROGRAM 2-6 Print Value of Selected Characters

```
/* Display the decimal value of selected characters,
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   char A         = 'A';
   char a         = 'a';
   char B         = 'B';
   char b         = 'b';
   char Zed       = 'Z';
   char zed       = 'z';
   char zero      = '0';
   char eight     = '8';
   char NL        = '\n';      // newline
   char HT        = '\t';      // horizontal tab
   char VT        = '\v';      // vertical tab
   char SP        = ' ';       // blank or space
   char BEL       = '\a';      // alert (bell)
   char dblQuote  = '"';       // double quote
   char backSlash = '\\';      // backslash itself
   char oneQuote  = '\'';      // single quote itself

// Statements
   printf("ASCII for char 'A'  is: %d\n",  A);
   printf("ASCII for char 'a'  is: %d\n",  a);
   printf("ASCII for char 'B'  is: %d\n",  B);
   printf("ASCII for char 'b'  is: %d\n",  b);
   printf("ASCII for char 'Z'  is: %d\n",  Zed);
   printf("ASCII for char 'z'  is: %d\n",  zed);
   printf("ASCII for char '0'  is: %d\n",  zero);
   printf("ASCII for char '8'  is: %d\n",  eight);
   printf("ASCII for char '\\n' is: %d\n", NL);
   printf("ASCII for char '\\t' is: %d\n", HT);
   printf("ASCII for char '\\v' is: %d\n", VT);
   printf("ASCII for char ' '  is: %d\n",  SP);
   printf("ASCII for char '\\a' is: %d\n", BEL);
   printf("ASCII for char '\"'  is: %d\n", dblQuote);
   printf("ASCII for char '\\'  is: %d\n", backSlash);
   printf("ASCII for char '\''  is: %d\n", oneQuote);
```

continued

PROGRAM 2-6 Print Value of Selected Characters *(continued)*

```
   return 0;
}  // main
```

```
Results:
ASCII for character 'A'  is: 65
ASCII for character 'a'  is: 97
ASCII for character 'B'  is: 66
ASCII for character 'b'  is: 98
ASCII for character 'Z'  is: 90
ASCII for character 'z'  is: 122
ASCII for character '0'  is: 48
ASCII for character '8'  is: 56
ASCII for character '\n' is: 10
ASCII for character '\t' is: 9
ASCII for character '\v' is: 11
ASCII for character ' '  is: 32
ASCII for character '\a' is: 7
ASCII for character '"'  is: 34
ASCII for character '\'  is: 92
ASCII for character '''  is: 39
```

EXAMPLE 2-5 Define Constants

Let's write a program that calculates the area and circumference of a circle using a preprocessor-defined constant for π. Although we haven't shown you how to make calculations in C, if you know algebra you will have no problem reading the code in Program 2-7.

PROGRAM 2-7 Calculate a Circle's Area and Circumference

```
/* This program calculates the area and circumference
   of a circle using PI as a defined constant.
      Written by:
      Date:
*/
#include <stdio.h>
#define PI  3.1416

int main (void)
{
// Local Declarations
   float circ;
   float area;
   float radius;
```

continued

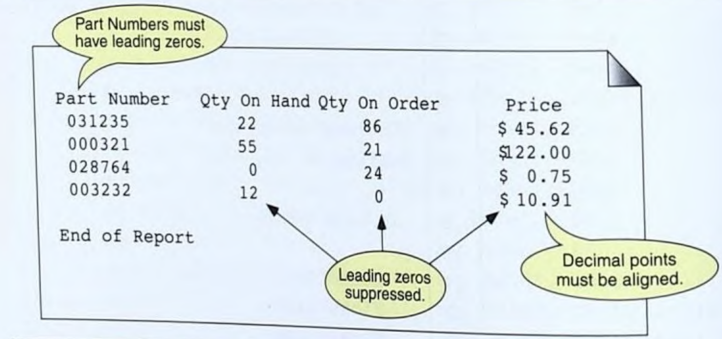
PROGRAM 2-7 Calculate a Circle's Area and Circumference *(continued)*

```

// Statements
   printf("\nPlease enter the value of the radius: ");
   scanf("%f", &radius);

   circ  = 2  * PI     * radius;
   area  = PI * radius * radius;

   printf("\nRadius is :         %10.2f", radius);
   printf("\nCircumference is : %10.2f", circ);
   printf("\nArea is :           %10.2f", area);

   return 0;
} // main
```

```
Results:
Please enter the value of the radius: 23

Radius is :               23.00
Circumference is :       144.51
Area is :               1661.91
```

EXAMPLE 2-6 Print a Report

You are assigned to a new project that is currently being designed. To give the customer an idea of what a proposed report might look like, the project leader has asked you to write a small program to print a sample. The specifications for the report are shown in Figure 2-22, and the code is shown in Program 2-8.



```
Part Number  Qty On Hand Qty On Order     Price
 031235          22           86         $ 45.62
 000321          55           21         $122.00
 028764           0           24         $  0.75
 003232          12            0         $ 10.91

End of Report
```

Part Numbers must have leading zeros.
Leading zeros suppressed.
Decimal points must be aligned.

FIGURE 2-22 Output Specifications for Inventory Report

The report contains four fields: a part number, which must be printed with leading zeros; the current quantity on hand; the current quantity on order; and the price of the item, printed to two decimal points. All data should be aligned in columns with captions indicating the type of data in each column. The report should be closed with an "End of Report" message.

PROGRAM 2-8 A Sample Inventory Report

```
/* This program will print four lines of inventory data
   on an inventory report to give the user an idea of
   what a new report will look like. Since this is not
   a real report, no input is required. The data are
   all specified as constants
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Statements
   // Print captions
   printf("\tPart Number\tQty On Hand");
   printf("\tQty On Order\tPrice\n");

   // Print data
   printf("\t %06d\t\t%7d\t\t%7d\t\t $%7.2f\n",
           31235, 22, 86, 45.62);
   printf("\t %06d\t\t%7d\t\t%7d\t\t $%7.2f\n",
           321, 55, 21, 122.);
   printf("\t %06d\t\t%7d\t\t%7d\t\t $%7.2f\n",
           28764, 0, 24, .75);
   printf("\t %06d\t\t%7d\t\t%7d\t\t $%7.2f\n",
           3232, 12, 0, 10.91);

   // Print end message
   printf("\n\tEnd of Report\n");
   return 0;
}  // main
```

Program 2-8 Analysis There are a few things about Program 2-8 that you should note. First, it is fully documented. Professional programmers often ignore documentation on "one-time-only" programs, thinking they will throw them away, only to find that they end up using them over and over. It only takes a few minutes to document a program, and it is always time well spent. If nothing else, it helps clarify the program in your mind.

Next, look carefully at the formatting for the print statements. Spacing is controlled by a combination of tabs and format code widths. The double spacing for the end of report message is controlled by placing a newline command (`\n`) at the beginning of the message in Statement 29.

Finally, note that the program concludes with a return statement that informs the operating system that it concluded successfully. Attention to details, even in small programs, is the sign of a good programmer.

EXAMPLE 2-7 Printing The Attributes of a Complex Number

A complex number is made of two components: a real part and an imaginary part. In mathematics, it can be represented as a vector with two components. The real part is the projection of the vector on the horizontal axis (x) and the imaginary part is the projection of the vector on the vertical axis (y). In C, we use complex number and a predefined library function to print the real and imaginary values. We can also find the length of the vector, which is the absolute value of the complex number and the angle of the vector, which is the argument of the vector. These four attributes are shown in Figure 2-23.

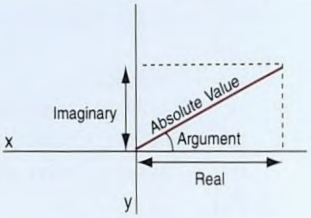


FIGURE 2-23 Complex Number Attributes

As the figure shows, the absolute value of the complex a + b * I can be found as $(a+b)^{1/2}$. The argument can be found as arctan (b/a). The conjugate of a complex number is another complex number defined as a – b * I.

Program 2-9 shows how we print the different attributes of a complex number using the predefined functions *creal*, *cimag*, *cabs*, and *carg*.

PROGRAM 2-9 Print Complex Number Attributes

```
/* Print attributes of a complex number.
      Written by:
      Date
*/
#include <stdio.h>
#include <math.h>
#include <complex.h>

int main (void)
```

continued

PROGRAM 2-9 Print Complex Number Attributes *(continued)*

```c
{
// Local Declarations
   double complex x = 4 + 4 * I;
   double complex xc;

// Statements
   xc = conj (x);
   printf("%f %f %f %f\n", creal(x), cimag(x),
                            cabs(x),  carg(x));

   printf("%f %f %f %f\n", creal(xc), cimag(xc),
                            cabs(xc),  carg(xc));
   return 0;
}  // main
```

```
Results:
4.000000 4.000000 5.656854 0.785398
4.000000 -4.000000 5.656854 -0.785398
```

EXAMPLE 2-8 Mathematics with Complex Numbers.

C allows us to add, subtract, multiply, and divide two complex numbers using the same operators (+, -, *, /) that we use for real numbers. Program 2-10 demonstrates the arithmetic use of operators with complex numbers.

PROGRAM 2-10 Complex Number Arithmetic

```c
/* Demonstrate complex number arithmetic.
      Written by:
      Date:
*/
#include <stdio.h>
#include <math.h>
#include <complex.h>

int main (void)
{
// Local Declarations
   double complex x = 3 + 4 * I;
   double complex y = 3 - 4 * I;
   double complex sum;
   double complex dif;
   double complex mul;
   double complex div;
```

continued

PROGRAM 2-10 Complex Number Arithmetic *(continued)*

```

// Statements
   sum = x + y;
   dif = x - y;
   mul = x * y;
   div = x / y;

   printf("%f %f %f %f\n", creal(sum), cimag(sum),
                          cabs(sum),  carg(sum));
   printf("%f %f %f %f\n", creal(dif), cimag(dif),
                          cabs(dif),  carg(dif));
   printf("%f %f %f %f\n", creal(mul), cimag(mul),
                          cabs(mul),  carg(mul));
   printf("%f %f %f %f\n", creal(div), cimag(div),
                          cabs(div),  carg(div));
   return 0;
} // main
```

```
Results:
6.000000 0.000000 6.000000 0.000000
0.000000 8.000000 8.000000 1.570796
25.000000 0.000000 25.000000 0.000000
-0.280000 0.960000 1.000000 1.854590
```

2.9 Software Engineering

Although this chapter introduces only a few programming concepts, there is still much to be said from a software engineering point of view. We will discuss the concepts of program documentation, data naming, and data hiding.

Program Documentation

There are two levels of program documentation. The first is the general documentation at the start of the program. The second level is found within each function.

General Documentation

Program 2-11 shows what we recommend for program documentation. Each program should start with a general description of the program. Following the general description is the name of the author and the date the program was written. Following the date is the program's change history, which documents the reason and authority for all changes. For a production program whose use spans several years, the change history can become extensive.

PROGRAM 2-11 Sample of General Program Documentation

```
/* A sample of program documentation. Each program
   starts with a general description of the program.
   Often, this description can be taken from the
   requirements specification given to the programmer.
     Written by: original author
     Date:       Date first released to production
   Change History:
     <date> Included in this documentation is a short
            description of each change.
*/
```

Module Documentation

Whenever necessary, we include a brief comment for blocks of code. A block of code is much like a paragraph in a report. It contains one thought—that is, one set of statements that accomplish a specific task. Blocks of code in our program are separated by blank program lines, just as we skip blank lines between paragraphs in reports.

If the block of code is difficult, or if the logic is especially significant, then we give the reader a short—one- or two-line—description of the block's purpose and/or operation. We will provide many examples of this type of documentation throughout the text.

Sometimes a textbook suggests that each variable in a program be documented. We disagree with this approach. First, the proper location for variable documentation is in a data dictionary. A data dictionary is a system documentation tool that contains standard names, descriptions, and other information about data used in a system.

Second, good data names eliminate the need for variable comments. In fact, if you think you need to document the purpose of a variable, check your variable name. You will usually find that improving the name eliminates the need for the comment.

Data Names

Another principle of good structured programming is the use of **intelligent data names**. This means that the variable name itself should give the reader a good idea about what data it contains and maybe even an idea about how the data are used.

Although there are obvious advantages to keeping names short, the advantage is quickly lost if the names become so cryptic that they are unintelligible. We have seen programmers struggle for hours to find a bug, only to discover that the problem was the wrong variable was used. The time saved keying short, cryptic names is often lost ten- or a hundredfold in debugging time.

We have formulated several guidelines to help you construct good, intelligent data names.

1. The name should match the terminology of the user as closely as possible.

   Let's suppose that you are writing a program to calculate the area of a rectangle. Mathematicians often label the sides of a rectangle *a* and *b*, but their real names are length and width. Therefore, your program should call the sides of the rectangle `length` and `width`. These names are commonly used by anyone describing a rectangle.

2. When necessary for readability, and to separate similar variables from each other, combine terms to form a variable name.

   Suppose that you are working on a project to compute a payroll. There are many different types of taxes. Each of the different taxes should be clearly distinguished from the others by good data names. Table 2-13 shows both good and bad names for this programming situation. Most of the poor names are either too abbreviated to be meaningful (such as `ftr`) or are generic names (such as `rate`) that could apply to many different pieces of data.

Good Names		Poor Names
ficaTaxRate	fica_tax_rate	rate ftr frate fica
ficaWitholding	fica_witholding	fwh ficaw wh
ficaWthldng	fica_wthldng	fcwthldng wthldng
ficaMax	ficaDlrMax	max fmax

TABLE 2-13 Examples of Good and Poor Data Names

Note the two different concepts for separating the words in a variable's name demonstrated in Table 2-13. In the first example, we capitalized the first letter of each word. In the second example, we separated the words with an underscore. Both are good techniques for making a compound name readable. If you use capitalization, keep in mind that C is case sensitive, so you must be careful to use the same cases for the name each time you use it.

3. Do not create variable names that are different by only one or two letters, especially if the differences are at the end of the word. Names that are too similar create confusion. On the other hand, a naming pattern makes it easier to recall the names. This is especially true when user terminology is being used. Thus, we see that the good names in Table 2-13 all start with `fica`.

4. Abbreviations, when used, should clearly indicate the word being abbreviated.

   Table 2-13 also contains several examples of good abbreviations. Whenever possible, use abbreviations created by the users. They will often have a glossary of abbreviations and acronyms that they use.

   Short words are usually not abbreviated. If they are short in the first place, they don't need to be made shorter.

5. Avoid the use of generic names.

   Generic names are programming or user jargon. For example, `count` and `sum` are both generic names. They tell you their purpose but don't give you any clue as to the type of data they are associated with. Better names would be `emplyCnt` and `ficaSum`. Programmers are especially fond of using generic names, but they tend to make the program confusing. Several of the poor names in Table 2-13 are generic.

6. Use memory constants or defined constants rather than literals for values that are hard to read or that might change from system to system.

   Some constants are nearly impossible to read. We pointed out the space earlier. If you need a space often, create a defined constant for it. Table 2-14 contains several examples of constants that are better when coded as defined constants.

#define SPACE ' '	#define BANG '!'
#define DBL_QTE '"'	#define QUOTE '\''
#define COMMA ','	#define COLON ':'

TABLE 2-14 Examples of Defined Constants

Data Hiding

In "Structure of a C Program" in Section 2.2, we discussed the concept of global and local variables. We pointed out that anything placed before *main* was said to be in the global part of the program. With the exception of data that must be visible to other programs, no variables should be placed in this section.

One of the principles of structured programming states that the data structure should be hidden from view. The two terms you usually hear in connection with this concept are **data hiding** and **data encapsulation**. Both of these principles have as their objective protecting data from accidental destruction by parts of your program that don't require access to the data. In other words, if a part of your program doesn't require data to do its job, it shouldn't be able to see or modify the data. Until you learn to use functions in Chapter 4, however, you will not be able to provide this data-hiding capability.

Nevertheless, you should start your programming with good practices. And since our ultimate objective is good structured programming, we now formulate our first programming standard:

Programming Standard

No variables are to be placed in the global area of a program.

Any variables placed in the global area of your program—that is, before *main*—can be used and changed by every part of your program. This is undesirable and is in direct conflict with the structured programming principles of data hiding and data encapsulation.

2.10 Tips and Common Programming Errors

1. Well-structured programs use global (defined) constants but do not use global variables.
2. The function header for *main* should be complete. We recommend the following format:

   ```
   int main (void)
   ```

   a. If you forget the parentheses after *main*, you will get a compile error.

   b. If you put a semicolon after the parentheses, you will get a compile error.

   c. If you misspell *main* you will not get a compile error, but you will get an error when you try to link the program. All programs must have a function named *main*.
3. If you forget to close the format string in the *scanf* or *printf* statement, you will get a compile error.
4. Using an incorrect conversion code for the data type being read or written is a run-time error. You can't read an integer with a *float* conversion code. Your program will compile with this error, but it won't run correctly.
5. Not separating read and write parameters with commas is a compile error.
6. Forgetting the comma after the format string in a read or write statement is a compile error.
7. Not terminating a block comment with a close token (`*/`) is a compile error.
8. Not including required libraries, such as *stdio.h*, at the beginning of your program is an error. Your program may compile, but the linker cannot find the required functions in the system library.
9. If you misspell the name of a function, you will get an error when you link the program. For example, if you misspell *scanf* or *printf*, your program will compile without errors, but you will get a linker error. Using the wrong case is a form of spelling error. For example, each of the following function names are different:

   ```
           scanf, Scanf, SCANF    printf, Printf, PRINTF
   ```
10. Forgetting the address operator (`&`) on a *scanf* parameter is a logic (run-time) error.
11. Do not use commas or other characters in the format string for a *scanf* statement. This will most likely lead to a run-time error when the user does not enter matching commas or characters. For example, the comma in the following statement will create a run-time problem if the user doesn't enter it exactly as coded.

    ```
    scanf ("%d, %d", &a, &b);
    ```

12. Unless you specifically want to read a whitespace character, put a space before the character conversion specification in a *scanf* statement.
13. Using an address operator (&) with a variable in the *printf* statement is usually a run-time error.
14. Do not put a trailing whitespace at the end of a format string in *scanf*. This is a fatal run-time error.

2.11 Key Terms

address list
address operator
ASCII
binary stream
block comment
Boolean
call
character constant
character set
comment
complex type
constant
conversion code
conversion specification
data encapsulation
data hiding
declaration
definition
derived types
end of file (EOF)
escape character
flag modifier
floating-point type
format control string
function
global declaration section
header file
identifier
imaginary type
include
initializer
integral type
intelligent data name
justification
keyword
Latin character set
line comment
literal
logical data
memory constant
padding
parameter
parameter list
precision modifier
program documentation
real type
reserved word
sign flag
size modifier
standard input
standard output
statement
standard types
statement section
stream
string
string constant
syntax
text stream
token
type
type qualifier
variable
width modifier

2.12 Summary

- In 1972, Dennis Ritchie designed C at Bell Laboratories.
- In 1989, the American National Standards Institute (ANSI) approved ANSI C; in 1990, the ISO standard was approved.
- The basic component of a C program is the function.
- Every C function is made of declarations, definitions, and one or more statements.
- One and only one of the functions in a C program must be called *main*.

- ❑ To make a program more readable, use comments. A comment is a sequence of characters ignored by the compiler. C uses two types of comments: block and line. A block comment starts with the token `/*` and ends with the token `*/`. A line comment starts with the `//` token; the rest of the line is ignored.
- ❑ Identifiers are used in a language to name objects.
- ❑ C types include *void*, integral, floating point, and derived.
- ❑ A *void* type is used when C needs to define a lack of data.
- ❑ An integral type in C is further divided into Boolean, character, and integer.
  - ■ A Boolean data type takes only two values: *true* and *false*. It is designated by the keyword *bool*.
  - ■ A character data type uses values from the standard alphabet of the language, such as ASCII or Unicode. There are two character type sizes, *char* and *w_char*.
  - ■ An integer data type is a number without a fraction. C uses four different integer sizes: *short int*, *int*, *long int*, and *long long int*.
- ❑ The floating-point type is further divided into real, imaginary, and complex.
  - ■ A real number is a number with a fraction. It has three sizes: *float*, *double*, and *long double*.
  - ■ The imaginary type represents the imaginary part of a complex number. It has three sizes, *float imaginary*, *double imaginary*, and *long double imaginary*.
  - ■ The complex type contains a real and an imaginary part. C uses three complex sizes: *float complex*, *double complex*, and *long double complex*.
- ❑ A constant is data whose value cannot be changed.
- ❑ Constants can be coded in three different ways: as literals, as define commands, and as memory constants.
- ❑ Variables are named areas of memory used to hold data.
- ❑ Variables must be declared and defined before being used in C.
- ❑ To input data through the keyboard and to output data through the monitor, use the standard formatted input/output functions.
- ❑ *scanf* is a standard input function for inputting formatted data through the keyboard.
- ❑ *printf* is a standard output function for outputting formatted data to the monitor.
- ❑ As necessary, programs should contain comments that provide the reader with in-line documentation for blocks of code.
- ❑ Programs that use "intelligent" names are easier to read and understand.

2.13 Practice Sets

Review Questions

1. The purpose of a header file, such as *stdio.h*, is to store a program's source code.

    a. True
    b. False

2. Any valid printable ASCII character can be used in an identifier.

    a. True
    b. False

3. The C standard function that receives data from the keyboard is *printf*.

    a. True
    b. False

4. Which of the following statements about the structure of a C program is false?

    a. A C program starts with a global declaration section.
    b. Declaration sections contain instructions to the computer.
    c. Every program must have at least one function.
    d. One and only one function may be named *main*.
    e. Within each function there is a local declaration section.

5. Which of the following statements about block comments is false?

    a. Comments are internal documentation for programmers.
    b. Comments are used by the preprocessor to help format the program.
    c. Comments begin with a `/*` token.
    d. Comments cannot be nested.
    e. Comments end with a `*/` token.

6. Which of the following identifiers is not valid?

    a. `_option`
    b. `amount`
    c. `sales_amount`
    d. `salesAmount`
    e. `$salesAmount`

7. Which of the following is not a data type?

    a. *char*
    b. *float*
    c. *int*
    d. *logical*
    e. *void*

8. The code that establishes the original value for a variable is known as a(n):

a. assignment
b. constant
c. initializer
d. originator
e. value

9. Which of the following statements about a constant is true?

a. Character constants are coded using double quotes (").
b. It is impossible to tell the computer that a constant should be a *float* or a *long double*.
c. Like variables, constants have a type and may be named.
d. Only integer values can be used in a constant.
e. The value of a constant may be changed during a program's execution.

10. The ______________ conversion specification is used to read or write a short integer.

a. `%c`
b. `%d`
c. `%f`
d. `%hd`
e. `%lf`

11. To print data left justified, you would use a ______________ in the conversion specification.

a. flag
b. precision
c. size
d. width
e. width and precision

12. The ______________ function reads data from the keyboard.

a. *displayf*
b. *printf*
c. *read*
d. *scanf*
e. *write*

13. One of the most common errors for new programmers is forgetting to use the address operator for variables in a *scanf* statement. What is the address operator?

a. The address modifier (`@`) in the conversion specification
b. The ampersand (`&`)
c. The caret (`^`)
d. The percent (`%`)
e. The pound sign (`#`)

Exercises

14. Which of the following is *not* a character constant in C?

a. `'C'`
b. `'bb'`
c. `"C"`
d. `'?'`
e. `' '`

15. Which of the following is *not* an integer constant in C?

a. `-320`
b. `+45`
c. `-31.80`
d. `1456`
e. `2,456`

16. Which of the following is *not* a floating-point constant in C?

a. `45.6`
b. `-14.05`
c. `'a'`
d. `pi`
e. `40`

17. What is the type of each of the following constants?

a. `15`
b. `-14.24`
c. `'b'`
d. `"1"`
e. `"16"`

18. Which of the following is *not* a valid identifier in C?

a. A3
b. `4A`
c. `if`
d. `IF`
e. `tax-rate`

19. What is the type of each of the following constants?

a. `"7"`
b. `3`
c. `"3.14159"`
d. `'2'`
e. `5.1`

20. What is the type of each of the following constants?

a. `"Hello"`
b. `15L`
c. `8.5L`
d. `8.5f`
e. `'\a'`

21. Which of the following identifiers are valid and which are invalid? Explain your answer.

    a. `num`

    b. `num2`

    c. `2dNum`

    d. `2d_num`

    e. `num#2`

22. Which of the following identifiers are valid and which are invalid? Explain your answer.

    a. `num-2`

    b. `num 2`

    c. `num_2`

    d. `_num2`

    e. `_num_2`

23. What is output from the following program fragment? To show your output, draw a grid of at least 8 lines with at least 15 characters per line.

```
// Local Declarations
int   x = 10;
char  w = 'Y';
float z = 5.1234;

// Statements
printf("\nFirst\nExample\n:");
printf("%5d\n, w is %c\n", x, w);
printf("\nz is %8.2f\n", z);
```

24. Find any errors in the following program.

```
// This program does nothing
int main
{
 return 0;
}
```

25. Find any errors in the following program.

```
#include (stdio.h)
int main (void)
{
    print ("Hello World");
    return 0;
{
```

26. Find any errors in the following program.

```
include <stdio>
int main (void)
{
 printf('We are to learn correct');
 printf('C language here');
 return 0;
} // main
```

27. Find any errors in the following program.

```
/* This is a program with some errors
   in it to be corrected.
*/
int main (void)
{
// Local Declarations
   integer        a;
   floating-point b;
   character      c;

 // Statements
   printf("The end of the program.");
   return 0;
 }  // main
```

28. Find any errors in the following program.

```
/* This is another program with some
   errors in it to be corrected.
*/
int main (void)
{
// Local Declarations
   a  int;
   b  float, double;
   c, d char;
 // Statements
   printf("The end of the program.");
   return 0;
 } // main
```

29. Find any errors in the following program.

```
/* This is the last program to be
   corrected in these exercises.
*/
```

continued

```
int main (void)
{
// Local Declarations
   a  int;
   b : c : d char;
   d , e, f double float;
// Statements
printf("The end of the program.");
return 0;
} // main
```

Problems

30. Code the variable declarations for each of the following:

a. a character variable named `option`

b. an integer variable, `sum`, initialized to `0`

c. a floating-point variable, `product`, initialized to 1

31. Code the variable declarations for each of the following:

a. a short integer variable named code

b. a constant named `salesTax` initialized to .0825

c. a floating-point named `sum` of size double initialized to `0`

32. Write a statement to print the following line. Assume the total value is contained in a variable named `cost`.

```
The sales total is: $    172.53
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

33. Write a program that uses four print statements to print the pattern of asterisks shown below.

```
******
******
******
******
```

34. Write a program that uses four print statements to print the pattern of asterisks shown below.

```
*
**
***
****
```

35. Write a program that uses defined constants for the vowels in the alphabet and memory constants for the even decimal digits (0, 2, 4, 6, 8). It then prints the following three lines using literal constants for the odd digits.

```
a e i o u
0 2 4 6 8
1 3 5 7 9
```

36. Write a program that defines five integer variables and initializes them to 1, 10, 100, 1000, and 10000. It then prints them on a single line separated by space characters using the decimal conversion code (%d), and on the next line with the float conversion code (%f). Note the differences between the results. How do you explain them?

37. Write a program that prompts the user to enter a quantity and a cost. The values are to be read into an integer named quantity and a float named unitPrice. Define the variables, and use only *one statement* to read the values. After reading the values, skip one line and print each value, with an appropriate name, on a separate line.

38. Write a program that prompts the user to enter an integer and then prints the integer first as a character, then as a decimal, and finally as a float. Use separate print statements. A sample run is shown below.

```
The number as a character: K
The number as a decimal  : 75
The number as a float    : 0.000000
```

Projects

39. Write a C program using *printf* statements to print the three first letters of your first name in big blocks. This program does not read anything from the keyboard. Each letter is formed using seven rows and five columns using the letter itself. For example, the letter *B* is formed using 17 B's, as shown below as part of the initials BEF.

```
BBB    EEEEE  FFFFF
B  B   E      F
B  B   E      F
BBB    EEE    FFF
B  B   E      F
B  B   E      F
BBB    EEEEE  F
```

This is just an example. Your program must print the first three letters of your first name. Design your *printf* statements carefully to create enough blank lines at the beginning and end to make your initials readable. Use comments in your program to enhance readability as shown in this chapter.

40. Write a program that reads a character, an integer, and a floating-point number. It then prints the character, first using a character format specification (`%c`) and then using an integer specification (`%d`). After printing the character, it prints the integer and floating-point numbers on separate lines. Be sure to provide complete instructions (prompts) for the user.

41. Write a program that prompts the user to enter three numbers and then prints them vertically (each on one line), first forward and then reversed (the last one first), as shown in the following design.

```
Please enter three numbers: 15 35 72
Your numbers forward:
   15
   35
   72

Your numbers reversed:
   72
   35
   15
```

42. Write a program that reads 10 integers and prints the first and the last on one line, the second and the ninth on the next line, the third and the seventh on the next line, and so forth. Sample input and the results are shown below.

```
Please enter 10 numbers:
10 31 2 73 24 65 6 87 18 9

Your numbers are:
   10   9
   31  18
    2  87
   73   6
   24  65
```

43. Write a program that reads nine integers and prints them three in a line separated by commas as shown below.

```
Input:
   10 31 2 73 24 65 6 87 18
Output
   10, 31,  2
   73, 24, 65
    6, 87, 18
```

Chapter 3
Structure of a C Program

Two features set the C language apart from many other languages: expressions and pointers. Both of these concepts lie at the very heart of the language, giving C its unique look and feel.

This chapter explores the first of these concepts: expressions. Expressions are not new to you; you have used them in mathematics. However, C's use of expressions is unique to the C language.

Closely tied to the concept of expressions are operators, precedence and associativity, and statements, all of which are discussed in this chapter. This chapter also introduces a concept known as side effects and explains in detail how it affects statements in C.

Objectives

❑ To be able to list and describe the six expression categories

❑ To understand the rules of precedence and associativity in evaluating expressions

❑ To understand the result of side effects in expression evaluation

❑ To be able to predict the results when an expression is evaluated

❑ To understand implicit and explicit type conversion

❑ To understand and use the first four statement types: null, expression, return, and compound

3.1 Expressions

An **expression** is a sequence of operands and operators that reduces to a single value. Expressions can be simple or complex. An **operator** is a syntactical token that requires an action be taken. An **operand** is an object on which an operation is performed; it receives an operator's action.

A **simple expression** contains only one operator. For example 2 + 5 is a simple expression whose value is 7; similarly, -a is a simple expression. A **complex expression** contains more that one operator. An example of a complex expression is 2 + 5 * 7. To evaluate a complex expression, we reduce it to a series of simple expressions. In the previous example, we first evaluate the simple expression 5 * 7 (35) and then the expression 2 + 35, giving a result of 37.

Every language has operators whose actions are clearly specified in the language syntax. The order in which the operators in a complex expression are evaluated is determined by a set of priorities known as **precedence**; the higher the precedence, the earlier the expression containing the operator is evaluated. The table inside the front cover contains the precedence of each operator. Looking at that table, we see that in the expression 2 + 5 * 7, multiplication has a higher priority than addition so the multiply expression is evaluated first. We discuss precedence in more detail in the next section.

If two operators with the same precedence occur in a complex expression, another attribute of an operator, its associativity, takes control. **Associativity** is the parsing direction used to evaluate an expression. It can be either left-to-right or right-to-left. When two operators with the same precedence occur in an expression and their associativity is left-to-right, the left operator is evaluated first. For example, in the expression 3 * 4 / 6, there are two operators, multiplication and division, with the same precedence and left-to-right associativity. Therefore, the multiplication is evaluated before the division. We also discuss associativity in more detail in the next section.

An expression always reduces to a single value.

We can divide simple expressions into six categories based on the number of operands, relative positions of the operand and operator, and the precedence of operator. Figure 3-1 shows the categories.

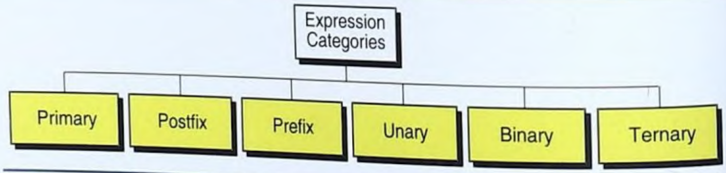


FIGURE 3-1 Expression Categories

Primary Expressions

The most elementary type of expression is a primary expression. A **primary expression** consists of only one operand with no operator. In C, the operand in the primary expression can be a name, a constant, or a parenthesized expression. Although a primary expression has no operator, the null operator in this expression has the precedence of 16 according to the table of precedence. In other words, a primary expression is evaluated first in a complex expression.

Names

A **name** is any identifier for a variable, a function, or any other object in the language. The following are examples of some names used as primary expressions:

```
a     b12     price     calc     INT_MAX     SIZE
```

Literal Constants

The second type of primary expression is the literal constant. As discussed in Chapter 2, a constant is a piece of data whose value can't change during the execution of the program. The following are examples of literal constants used as primary expressions:

```
5    123.98    'A'    "Welcome"
```

Parenthetical Expressions

The third type of primary expression is the parenthetical expression. Any value enclosed in parentheses must be reducible to a single value and is therefore a primary expression. This includes any of the complex expressions when they are enclosed in parentheses. Thus, a complex expression can be enclosed in parentheses to make it a primary expression. The following are primary expressions:

```
(2  *  3  +  4)      (a  =  23  +  b  *  6)
```

Postfix Expressions

The **postfix expression** consists of one operand followed by one operator. Its category is shown in Figure 3-2. There are several operators that create a postfix expression as you can see in the precedence table. We discuss only three of them here: function call, postfix increment, and postfix decrement.

Operand
Operator

FIGURE 3-2 Postfix Expressions

Function Call

We have already used a postfix expression. In the hello world program, we wrote a message on the monitor using the *printf* function. Function calls are postfix expressions. The function name is the operand and the operator is the parentheses that follow the name. The parentheses may contain arguments or be empty. When present, the arguments are part of the operator.

Postfix Increment/Decrement

The postfix increment and postfix decrement are also postfix operators. Virtually all programs require somewhere in their code that the value 1 be added to a variable. In early languages, this additive operation could only be represented as a binary expression. C provides the same functionality in two expressions: postfix and prefix.

In the postfix increment, the variable is increased by 1. Thus, a++ results in the variable a being increased by 1. The effect is the same as a = a + 1.

(a++) has the same effect as (a = a + 1)

Although both the postfix increment (a++) and binary expression (a = a + 1) add 1 to the variable, there is a major difference. The *value* of the postfix increment expression is determined *before* the variable is increased. For instance, if the variable a contains 4 before the expression is evaluated, the value of the expression a++ is 4. As a result of evaluating the expression and its side effect, a contains 5. The value and side effect of the postfix increment are graphically shown in Figure 3-3.

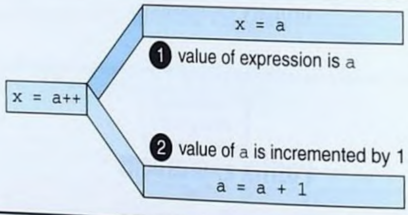


FIGURE 3-3 Result of Postfix a++

The postfix decrement (a--) also has a value and a side effect. As with the increment, the value of the expression is the value of a before the decrement; the side effect is the variable is decremented by 1.

The operand in a postfix expression must be a variable.

Program 3-1 demonstrates the effect of the postfix increment expression.

PROGRAM 3-1 Demonstrate Postfix Increment

```
/* Example of postfix increment.
      Written by:
      Date:
*/
#include <stdio.h>
int main (void)
{
// Local Declarations
   int a;

// Statements
   a = 4;
   printf("value of a     : %2d\n", a);
   printf("value of a++   : %2d\n",    a++);
   printf("new value of a: %2d\n\n", a);
   return 0;
}  // main
```

```
Results:
value of a     :  4
value of a++   :  4
new value of a:  5
```

Prefix Expressions

In prefix expressions, the operator comes before the operand as seen in Figure 3-4.

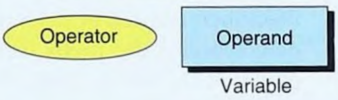


FIGURE 3-4 Prefix Expression

Prefix Increment/Decrement

In C, we have only two prefix operators that form prefix expressions: prefix increment and prefix decrement. Just like the postfix increment and postfix decrement operators, the prefix increment and prefix decrement operators are shorthand notations for adding or subtracting 1 from a variable.

The operand of a prefix expression must be a variable.

There is one major difference between the postfix and prefix operators, however: with the prefix operators, the effect takes place *before* the expression that contains the operator is evaluated. Note that this is the reverse of the postfix operation. Figure 3-5 shows the operation graphically.

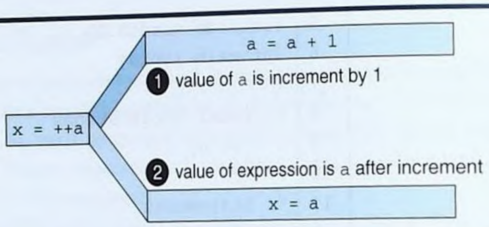

a = a + 1
1 value of a is increment by 1
x = ++a
2 value of expression is a after increment
x = a

FIGURE 3-5 Result of Prefix ++a

The effect of both the postfix and prefix increment is the same: The variable is incremented by 1. If we don't need the value of the expression—that is, if all we need is the effect of incrementing the value of a variable by 1—then it makes no difference which one we use. You will find that programmers use the postfix increment and decrement more often, if for no other reason than that the variable is shown first and is therefore easier to read.

(++a) has the same effect as (a = a + 1)

On the other hand, if we require both the value and the effect, then our application determines which one we need to use. When we need the value of the expression to be the *current value* of the variable, we use the postfix operator; when we need the value to be the *new value* of the variable (after it has been incremented or decremented), we use the prefix operator. Program 3-2 demonstrates the prefix increment expression. Study it carefully, and compare it to the results from Program 3-1.

PROGRAM 3-2 Demonstrate Prefix Increment

```
/* Example of prefix increment.
      Written by:
```

continued

PROGRAM 3-2 Demonstrate Prefix Increment *(continued)*

```
    Date:
*/
#include <stdio.h>
int main (void)
{
// Local Declarations
  int a;

// Statements
  a = 4;
  printf("value of a    : %2d\n", a);
  printf("value of ++a  : %2d\n", ++a);
  printf("new value of a: %2d\n", a);
  return 0;
}  // main
```

```
Results:
value of a    :  4
value of ++a  :  5
new value of a:  5
```

Program 3-2 Analysis The only difference in the printouts between Program 3-1 and Program 3-2 is the use of the increment operators. The first program uses the postfix increment; the second uses the unary prefix increment. In both cases, we start with the same value for a and it has the same value at the end. But the value of the expression itself is different. To help remember the difference, use this rule: If the ++ is *before* the operand, the increment takes place *before* the expression is evaluated; if it is *after* the operand, the increment takes place *after* the expression is evaluated.

If ++ is after the operand, as in a++, the increment takes place after the expression is evaluated. If ++ is before the operand, as in ++a, the increment takes place before the expression is evaluated.

Unary Expressions

A **unary expression**, like a prefix expression, consist of one operator and one operand. Also like the prefix expression, the operator comes before the operand. Although prefix expressions and unary expressions look the same, they belong to different expression categories because the prefix expression needs a variable as the operand while the unary expression can have an expression or a variable as the operand. Many of the unary expressions are also familiar to you from mathematics and will require little explanation. In this chapter we discuss the *sizeof* operator, the plus/minus operators, and the cast operator. The others will be discussed in later chapters. The format of the unary expressions is demonstrated in Figure 3-6.

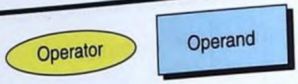


FIGURE 3-6 Unary Expressions

sizeof

The sizeof operator tells us the size, in bytes, of a type or a primary expression. By specifying the size of an object during execution, we make our program more portable to other hardware. A simple example will illustrate the point. On some personal computers, the size of the integer type is 2 bytes. On some mainframe computers, it is 4 bytes. On the very large supercomputers, it can be as large as 16 bytes. If it is important to know the exact size (in bytes) of an integer, we can use the *sizeof* operator with the integer type as shown below.

```
sizeof (int)
```

It is also possible to find the size of a primary expression. Here are two examples.

```
sizeof -345.23     sizeof x
```

Unary Plus/Minus

The unary plus and unary minus operators are what we think of as simply the plus and minus signs. In C, however, they are actually operators. Because they are operators, they can be used to compute the arithmetic value of an operand.

The plus operator does not change the value of the expression. If the expression's value is negative, it remains negative; if the expression's value is positive, it remains positive.

The minus operator changes the sign of a value algebraically—that is, to change it from plus to minus or minus to plus. Note, however, that the value of the stored variable is unchanged. The operation of these operators is seen in Table 3-1.

Expression	Contents of a Before *and* After Expression	Expression Value
+a	3	+3
-a	3	-3

continued

TABLE 3-1 Examples of Unary Plus And Minus Expressions

Expression	Contents of a Before *and* After Expression	Expression Value
+a	-5	-5
-a	-5	+5

TABLE 3-1 Examples of Unary Plus And Minus Expressions *(continued)*

Cast Operator

The third unary operator we discuss in this chapter is cast. The cast operator converts one expression type to another. For example, to convert an integer to a real number, we would use the following unary expression.

```
(float) x
```

It is important to note that, as with all of the unary operators, only the expression value is changed. The integer variable, x, is unchanged. We discuss the cast operator in detail in "Explicit Type Conversion (Cast)" in Section 3.5.

Binary Expressions

Binary expressions are formed by an operand-operator-operand combination. They are perhaps the most common expression category. Any two numbers added, subtracted, multiplied, or divided are usually formed in algebraic notation, which is a binary expression. There are many binary expressions. We cover the first two in this chapter. Figure 3-7 shows the format of a binary expression.

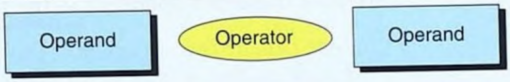


FIGURE 3-7 Binary Expressions

Multiplicative Expressions

The first binary expression we study, **multiplicative expressions**, which take its name from the first operator, include the multiply, divide, and modulus operators. These operators have the highest priority (13) among the binary operators and are therefore evaluated first among them.

The result of a multiply operator (*) is the product of the two operands. The operands can be any arithmetic type (integral or floating-point). The type of the result depends on the conversion rule that we discuss later in the chapter.

```
10 * 3                     // evaluates to 30
true * 4                   // evaluates to 4
'A' * 2                    // evaluates to 130
22.3  * 2                  // evaluates to 44.6
```

The result of a divide operator (/) depends on the type of the operands. If one or both operands is a floating-point type, the result is a floating-point quotient. If both operands are integral type, the result is the integral part of the quotient. The following shows some examples of the division operator.

```
10 / 3                       // evaluates to 3
true / 4                     // evaluates to 0
'A' / 2                      // evaluates to 32
22.3  / 2                    // evaluates to 11.15
```

Multiply and divide are well known, but you may not be familiar with the modulus operator (%), more commonly known as modulo. This operator divides the first operand by the second and returns the remainder rather than the quotient. Both operands must be integral types and the operator returns the remainder as an integer type. The following examples demonstrate the modulo operator.

```
10 % 3          // evaluates to 1
true % 4        // evaluates to 1
'A' % 10        // evaluates to 5
22.3  % 2       // Error: Modulo cannot be floating-point
```

Because the division and modulus operators are related, they are often confused. Remember: The value of an expression with the division operator is the quotient; the value of a modulus operator is the remainder. Study the effect of these two operators in the following expressions:

```
3 / 5                             // evaluates to 0
3 % 5                             // evaluates to 3
```

Another important point to remember is, if the first integral operand is smaller than the second integral operand, and the result of division is 0, the result of the modulo operator is the first operand as shown below:

```
3 / 7                             // evaluates to 0
3 % 7                             // evaluates to 3
```

Both operands of the modulo operator (%) must be integral types.

Additive Expressions

In **additive expressions**, the second operand is added to or subtracted from the first operand, depending on the operator used. The operands in an additive expression can be any arithmetic types (integral or floating-point). Additive operators have lower precedence (12) than multiplicative operators (13); therefore, they are evaluated after multiplicative expressions. Two simple examples are shown below:

```
3 + 7                                            // evaluates to 10
3 - 7                                            // evaluates to -4
```

EXAMPLE 3-1 Binary Expressions

Now let's look at a short program that uses some of these expressions. Program 3-3 contains several binary expressions.

PROGRAM 3-3 Binary Expressions

```
/* This program demonstrates binary expressions.
      Written by:
      Date:
*/
#include <stdio.h>
int main (void)
{
// Local Declarations
   int   a = 17;
   int   b = 5;
   float x = 17.67;
   float y = 5.1;

// Statements
   printf("Integral calculations\n");
   printf("%d + %d  = %d\n", a, b, a + b);
   printf("%d - %d  = %d\n", a, b, a - b);
   printf("%d * %d  = %d\n", a, b, a * b);
   printf("%d / %d  = %d\n", a, b, a / b);
   printf("%d %% %d  = %d\n", a, b, a % b);
   printf("\n");

   printf("Floating-point calculations\n");
   printf("%f + %f  = %f\n", x, y, x + y);
   printf("%f - %f  = %f\n", x, y, x - y);
   printf("%f * %f  = %f\n", x, y, x * y);
   printf("%f / %f  = %f\n", x, y, x / y);
```

continued

PROGRAM 3-3 Binary Expressions *(continued)*

```
28       return 0;
29    } // main
```

```
Results:
Integral calculations
17 + 5 = 22
17 - 5 = 12
17 * 5 = 85
17 / 5 = 3
17 % 5 = 2

Floating-point calculations
17.670000 + 5.100000  = 22.770000
17.670000 - 5.100000  = 12.570000
17.670000 * 5.100000  = 90.116997
17.670000 / 5.100000  = 3.464706
```

Program 3-3 Analysis This simple program requires only three explanatory comments. (1) Note that even for a simple program we include all of the standard documentation comments. (2) We do not recommend that you include calculations in print statements as we have done in this program—it is not a good structured programming technique. We include them in this program because we haven't yet shown you how to save the results of a calculation. (3) Study the format string in statement 20. To print a percent sign as text in the format string, we need to code two percent signs.

Assignment Expressions

The **assignment expression** evaluates the operand on the right side of the operator (=) and places its value in the variable on the left. The assignment expression has a value and a side effect.

- The value of the total expression is the value of the expression on the right of the assignment operator (=).
- The side effect places the expression value in the variable on the left of the assignment operator.

The left operand in an assignment expression must be a single variable.

There are two forms of assignment: simple and compound.

Simple Assignment

Simple assignment is found in algebraic expressions. Three examples of simple assignments are shown below.

```
a = 5          b = x + 1          i = i + 1
```

Of course, for the effect to take place, the left variable must be able to receive it; that is, it must be a variable, not a constant. If the left operand cannot receive a value and we assign one to it, we get a compile error.

Compound Assignment

A compound assignment is a shorthand notation for a simple assignment. It requires that the left operand be repeated as a part of the right expression. Five compound assignment operators are discussed in this chapter: `*=`, `/=`, `%=`, `+=`, and `-=`.

To evaluate a compound assignment expression, first change it to a simple assignment, as shown in Table 3-2. Then perform the operation to determine the value of the expression.

Compound Expression	Equivalent Simple Expression
`x *= expression`	`x = x * expression`
`x /= expression`	`x = x / expression`
`x %= expression`	`x = x % expression`
`x += expression`	`x = x + expression`
`x -= expression`	`x = x - expression`

TABLE 3-2 Expansion of Compound Expressions

When a compound assignment is used with an expression, the expression is evaluated first. Thus, the expression

```
x *= y + 3
```

is evaluated as

```
x = x * (y + 3)
```

which, given the values x is 10 and y is 5, evaluates to 80.

Program 3-4 demonstrates the first three compound expressions.

PROGRAM 3-4 Demonstration of Compound Assignments

```
/* Demonstrate examples of compound assignments.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
```

continued

PROGRAM 3-4 Demonstration of Compound Assignments *(continued)*

```
// Local Declarations
   int x;
   int y;

// Statements
   x = 10;
   y = 5;

   printf("x: %2d  |  y: %2d ",    x, y);
   printf("  |  x *= y + 2: %2d ", x *= y + 2);
   printf("  |  x is now: %2d\n",  x);

   x = 10;
   printf("x: %2d  |  y: %2d ",    x, y);
   printf("  |  x /= y + 1: %2d ", x /= y + 1);
   printf("  |  x is now: %2d\n",  x);

   x = 10;
   printf("x: %2d  |  y: %2d ",     x, y);
   printf("  |  x %%= y - 3: %2d ", x %= y - 3);
   printf("  |  x is now: %2d\n", x);

   return 0;
}  // main
```

```
Results:
x: 10  |  y:  5  |  x *= y + 2: 70  |  x is now: 70
x: 10  |  y:  5  |  x /= y + 1:  1  |  x is now:  1
x: 10  |  y:  5  |  x %= y - 3:  0  |  x is now:  0
```

Program 3-4 Analysis Note that we have used an assignment statement in the *printf* statements to demonstrate that an assignment expression has a value. As we said before, this is not good programming style, but we use it here to match the format we used in Program 3-3. *Do not hide calculations in print statements.* Also, since we are changing the value of x with each assignment, even though it is in a *printf* statement, we need to reset it to 10 for each of the print series.

3.2 Precedence and Associativity

Precedence is used to determine the order in which different operators in a complex expression are evaluated. **Associativity** is used to determine the order in which operators with the same precedence are evaluated in a complex expression. Another way of stating this is that associativity determines how operators *with the same precedence* are grouped together to form complex expressions. Precedence is applied before associativity to determine the order in which expressions are evaluated. Associativity is then applied, if necessary.

Precedence

The concept of precedence is well founded in mathematics. For example, in algebra, multiplication and division are performed before addition and subtraction. C extends the concept to 16 levels, as shown in the Precedence Table inside the front cover.

The following is a simple example of precedence:

```
2 + 3 * 4
```

This expression is actually two binary expressions, with one addition and one multiplication operator. Addition has a precedence of 12. Multiplication has a precedence of 13. This results in the multiplication being done first, followed by the addition, as shown below in the same expression with the default parentheses added. The value of the complete expression is 14.

```
(2 + (3 * 4)) → 14
```

As another example consider the following expression:

```
-b++
```

Two different operators are in this expression. The first is the unary minus, the second is the postfix increment. The postfix increment has the higher precedence (16), so it is evaluated first. Then the unary minus, with a precedence of 15, is evaluated. To reflect the precedence, we have recoded the expression using parentheses.

```
(-(b++))
```

Assuming that the value of `b` is 5 initially, the expression is evaluated to −5. What is the value of `b` after the expression is complete? It is 6 because the operator has an effect that is separate from the value of the expression.

Program 3-5 demonstrates precedence by printing the same expression, once without parentheses and once with parentheses to change the precedence. Because the parentheses create a primary expression that must be evaluated before the binary multiply, the answer is different.

PROGRAM 3-5 Precedence

```
/* Examine the effect of precedence on an expression.
      Written by:
```

continued

PROGRAM 3-5 Precedence *(continued)*

```
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int a = 10;
   int b = 20;
   int c = 30;

// Statements
   printf ("a *  b + c  is: %d\n", a *  b + c);
   printf ("a * (b + c) is: %d\n", a * (b + c));
   return 0;
}  // main
```

```
Results:
a *  b + c  is: 230
a * (b + c) is: 500
```

Associativity

Associativity can be left-to-right or right-to-left. **Left-to-right associativity** evaluates the expression by starting on the left and moving to the right. Conversely, **right-to-left associativity** evaluates the expression by proceeding from the right to the left. Remember, however, that associativity is used only when the operators all have the same precedence.

Associativity is applied when we have more than one operator of the same precedence level in an expression.

ASSOCIATIVITY

Left-to-right Associativity

The following shows an example of left-to-right associativity. Here we have four operators of the same precedence (`* / % *`).

```
3 * 8 / 4 % 4 * 5
```

Associativity determines how the subexpressions are grouped together. All of these operators have the same precedence (13). Their associativity is from left to right. So they are grouped as follows:

```
((((3 * 8) / 4) % 4) * 5)
```

The value of this expression is 10. A graphical representation of this expression is shown in Figure 3-8.

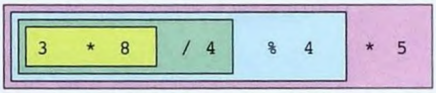


FIGURE 3-8 Left-to-right Associativity

Right-to-left Associativity

Several operators have right-to-left associativity as shown in the precedence table. For example, when more than one assignment operator occurs in an assignment expression, the assignment operators must be interpreted from right to left. This means that the rightmost expression will be evaluated first; then its value will be assigned to the operand on the left of the assignment operator and the next expression will be evaluated. Under these rules, the expression

```
a += b *= c -= 5
```

is evaluated as

```
(a += (b *= (c -= 5)))
```

which is expanded to

```
(a = a + (b = b * (c = c - 5)))
```

If a has an initial value of 3, b has an initial value of 5, and c has an initial value of 8, these expressions become

```
(a = 3 + (b = (5 * (c =  8 - 5)))
```

which results in c being assigned a value of 3, b being assigned a value of 15, and a being assigned a value of 18. The value of the complete expression is also 18. A diagram of this expression is shown in Figure 3-9.

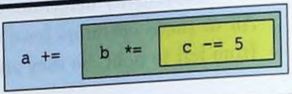


FIGURE 3-9 Right-to-left Associativity

A simple but common form of assignment is shown below. Suppose we have several variables that all need to be initialized to zero. Rather than initializing each separately, we can use a complex statement to do it.

```
a = b = c = d = 0;
```

3.3 Side Effects

A **side effect** is an action that results from the evaluation of an expression. For example, in an assignment, C first evaluates the expression on the right of the assignment operator and then places the value in the left variable. Changing the value of the left variable is a side effect. Consider the following expression:

```
x = 4;
```

This simple expression has three parts. First, on the right of the assignment operator is a primary expression that has the value 4. Second, the whole expression (x = 4) also has a value of 4. And third, as a side effect, x receives the value 4.

Let's modify the expression slightly and see the same three parts.

```
x = x + 4;
```

Assuming that x has an initial value of 3, the value of the expression on the right of the assignment operator has a value of 7. The whole expression also has a value of 7. And as a side effect, x receives the value of 7. To prove these three steps to yourself, write and run the following code fragment:

```
int x = 3;
printf("Step 1--Value of x: %d\n", x);
printf("Step 2--Value of x = x + 4: %d\n", x = x + 4);
printf("Step 3--Value of x now: %d\n", x);
```

Now, let's consider the side effect in the postfix increment expression. This expression is typically coded as shown below.

```
a++
```

As we saw earlier, the value of this expression is the value of a before the expression is evaluated. As a side effect, however, the value of a is incremented by 1.

In C, six operators generate side effects: prefix increment and decrement, postfix increment and decrement, assignment, and function call.

3.4 Evaluating Expressions

Now that we have introduced the concepts of precedence, associativity, and side effects, let's work through some examples.

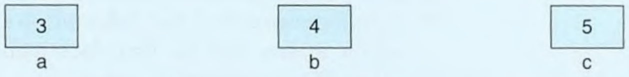
Expressions without Side Effects

EXAMPLE 3-2 Expression with no Side Effects

The first expression is shown below. It has no side effects, so the values of all of its variables are unchanged.

```
a * 4 + b / 2 - c * b
```

For this example, assume that the values of the variables are



To evaluate an expression *without side effects,* follow the simple rules shown below.

1. Replace the variables by their values. This gives us the following expression:

```
3 * 4 + 4 / 2 - 5 * 4
```

2. Evaluate the highest precedence operators, and replace them with the resulting value. In the above expression, the operators with the highest precedence are the multiply and divide (13). We therefore evaluate them first from the left and replace them with the resulting values. The expression is now

```
(3 * 4) + (4 / 2) - (5 * 4) → 12 + 2 - 20
```

3. Repeat step 2 until the result is a single value.

In this example, there is only one more precedence, binary addition and subtraction. After they are evaluated, the final value is –6. Since this expression had no side effects, all of the variables have the same values after the expression has been evaluated that they had at the beginning.

Expressions with Side Effects

EXAMPLE 3-3 Example with Side Effects

Now let's look at the rules for an expression that has side effects and parenthesized expressions. For this example, consider the expression

```
--a * (3 + b) / 2 - c++ * b
```

Assume that the variables have the values used above, a is 3, b is 4, c is 5. To evaluate this expression, use the following rules:

1. Calculate the value of the parenthesized expression (3 + b) first (precedence 16). The expression now reads

```
--a * 7 / 2 - c++ * b
```

2. Evaluate the postfix expression (c++) next (precedence 16). Remember that as a postfix expression, the value of c++ is the same as the value of c; the increment takes place after the evaluation. The expression is now

```
--a * 7 / 2 - 5 * b
```

3. Evaluate the prefix expression (--a) next (priority 15). Remember that as a prefix expression, the value of --a is the value after the side effect, which means that we first decrement a and then use its decremented value. The expression is now

```
2 * 7 / 2 - 5 * b
```

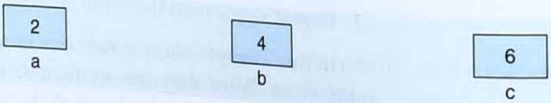
4. The multiply and division are now evaluated using their associativity rule, left to right, as shown below.

```
14 / 2 - 5 * b → 7 - 5 * 4 → 7 - 20
```

5. The last step is to evaluate the subtraction. The final expression value is -13 as shown in the final example.

```
7 - 20 → -13
```

After the side effects, the variables have the values shown below.



Program 3-6 evaluates the two expressions in this section.

PROGRAM 3-6 Evaluating Expressions

```
/* Evaluate two complex expressions.
      Written by:
      Date:
*/
#include <stdio.h>
int main (void)
{
// Local Declarations
   int a = 3;
   int b = 4;
   int c = 5;
   int x;
   int y;

// Statements
   printf("Initial values of the variables: \n");
   printf("a = %d\tb = %d\tc = %d\n\n", a, b, c);

   x = a * 4 + b / 2 - c * b;
   printf
     ("Value of  a *  4 + b  / 2 - c  * b:  %d\n", x);

   y = --a * (3 + b) / 2 - c++ * b;
   printf
     ("Value of --a * (3 + b) / 2 - c++ * b: %d\n", y);
   printf("\nValues of the variables are now: \n");
   printf("a = %d\tb = %d\tc = %d\n\n", a, b, c);

   return 0;
}  // main
```

```
Results:
Initial values of the variables:
a = 3   b = 4   c = 5

Value of   a *  4 + b  / 2 - c   * b:  -6
Value of --a * (3 + b) / 2 - c++ * b: -13

Values of the variables are now:
a = 2   b = 4   c = 6
```

Warning

A warning is in order: In C, if an expression variable is modified more than once during its evaluation, the result is undefined. C has no specific rule to cover when the side effect takes place, and compiler writers can implement the side effect in different ways. The result is that different compilers will give different expression results.

In C, if a single variable is modified more than once in an expression, the result is undefined.

WARNING

3.5 Type Conversion

Up to this point, we have assumed that all of our expressions involved data of the same type. But, what happens when we write an expression that involves two different data types, such as multiplying an integer and a floating-point number? To perform these evaluations, one of the types must be converted.

Implicit Type Conversion

When the types of the two operands in a binary expression are different, C automatically converts one type to another. This is known as **implicit type conversion**. For implicit type conversion, C has several complicated rules that we gradually introduce throughout the book. We mention some of the simple conversions in this section.

Conversion Rank

Before we discuss how conversions are handled, we need to discuss the concept of **conversion rank**. In C, we can assign a rank to the integral and floating-point arithmetic types. Figure 3-10 shows the ranks as we use them for conversion in this chapter. While the 1 to 9 scale we use is conceptually correct, the actual implementation is much more complex.

As shown in Figure 3-10, a *long double* real has a higher rank than a *long* integer and a short integer has a higher rank than a character.

Conversions in Assignment Expressions

A simple assignment involves an assignment operator and two operands. Depending on the difference in the rank, C tries to either promote or demote the right expression to make it the same rank as the left variable. **Promotion** occurs if the right expression has lower rank; **demotion** occurs if the right expression has a higher rank.

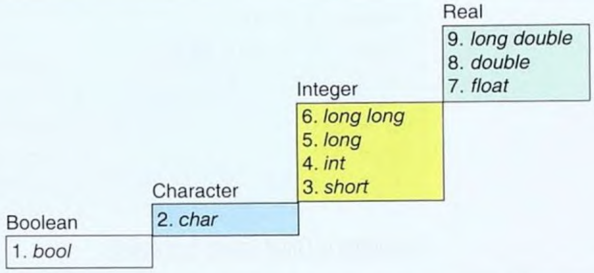


FIGURE 3-10 Conversion Rank

Promotion

There is normally no problem with promotion. The rank of the right expression is elevated to the rank of the left variable. The value of the expression is the value of the right expression after the promotion. The following examples demonstrate some simple promotions.

```
bool        b = true;
char        c = 'A';
int         i = 1234;
long double d = 3458.0004;
c = b;                         // value of c is SOH (ASCII 1)
i = c;                         // value of i is 65
d = b;                         // value of d is 1.0
d = i;                         // value of d is 1234.0
```

Demotion

Demotion may or may not create problems. If the size of the variable at the left side can accommodate the value of the expression, there is no problem; however, some of the results may surprise you.

Any integral or real value can be assigned to a Boolean type. If the value of the expression on the right is zero, *false* (0) is stored; if the result is not zero, either positive or negative, *true* (1) is stored.

When an integer or a real is assigned to a variable of type character, the least significant byte of the number is converted to a character and stored. When a real is stored in an integer, the fraction part is dropped. However, if the integral part is larger than the maximum value that can be stored, the results are invalid and unpredictable. Similarly, when we try to store a *long double* in a variable of type *float*, the results are valid if the value fits or invalid if it is too large.

The following examples demonstrate demotion.

```
bool  b = false;
char  c = 'A';
short s = 78;
int   j = INT_MAX;
int   k = 65;
...
b = c;                       // value of b is 1 (true)
s = j;                       // value of s is unpredictable
c = k + 1;                   // demotion: value of c is 'B'
```

Conversion in Other Binary Expressions

Conversion has a different set of rules for the other binary expressions. The rules are sometimes very complicated, but we can summarize them in three steps, which cover most cases:

1. The operand with the higher rank is determined using the ranking in Figure 3-10.
2. The lower-ranked operand is promoted to the rank defined in step 1. After the promotion, both expressions have the same rank.
3. The operation is performed with the expression value having the type of the promoted rank.

The following examples demonstrate some common conversions.

```
bool        b = true;
char        c = 'A';
int         i = 3650;
short       s = 78;
long double d = 3458.0004;
b + c                  // b promoted; result is 'B' ('A' + 1)
i * s;                 // result is an int
d * c;                 // result is long double
```

Let's look at a small program to see the effect of implicit conversions. In Program 3-7 we add a character, an integer, and a *float*. We can add characters to integers and floating-point values because all characters have an ASCII value that can be promoted.

PROGRAM 3-7 Implicit Type Conversion

```
/* Demonstrate automatic promotion of numeric types.
      Written by:
      Date:
*/
#include <stdio.h>
```

continued

PROGRAM 3-7 Implicit Type Conversion *(continued)*

```
 6  #include <stdbool.h>
 7
 8  int main (void)
 9  {
10  // Local Declarations
11     bool  b  = true;
12     char  c  = 'A';
13     float d  = 245.3;
14     int   i  = 3650;
15     short s  = 78;
16
17  // Statements
18     printf("bool + char is char:   %c\n", b + c);
19     printf("int * short is int:    %d\n", i * s);
20     printf("float * char is float: %f\n", d * c);
21
22     c = c + b;                       // bool promoted to char
23     d = d + c;                       // char promoted to float
24     b = false;
25     b = -d;                          // float demoted to bool
26
27     printf("\nAfter execution...\n");
28     printf("char + true:   %c\n", c);
29     printf("float + char:  %f\n", d);
30     printf("bool = -float:  %f\n", b);
31
32     return 0;
33  }  // main
```

```
Results:
bool + char is char:   B
int * short is int:    284700
float * char is float: 15944.500000

After execution...
char + true:   B
float + char:  311.299988
bool = -float:  1
```

Program 3-7 Analysis

Several points in this program require explanation. First, as we stated before, it is not a good programming practice to code an expression in a print statement. We do it in this program, however, to demonstrate the promotion or demotion of the expression.

The first print series displays the value of mixed type expressions. As you examine each result, note that the value printed is in the form of the higher ranked variable. For example, in statement 18 the Boolean in the expression (b) is promoted to a character and then added to the value of the character expression (c). The result is the character B, which is then passed to the *printf* function where it is printed using the format specification %c.

The second print series displays the results of assignments. In the first one, we add *true* (1) to the letter A. The result is B as you would expect. In the second assignment, we add the letter B from the previous assignment to a real number. The new value of the real number is almost 66 greater than the original value. The difference occurs because real numbers are not exact. Rather than storing 245.3, the value stored was 245.299988.

Finally, note what happens when we assign a negative, real number to a boolean. The result is *true*.

Explicit Type Conversion (Cast)

Rather than let the compiler implicitly convert data, we can convert data from one type to another ourself using **explicit type conversion**. Explicit type conversion uses the unary **cast** operator, which has a precedence of 14. To cast data from one type to another, we specify the new type in parentheses before the value we want converted. For example, to convert an integer, a, to a *float*, we code the expression shown below.

```
(float) a
```

Note that in this operation, like any other unary operation, the value stored in a is still of type *int*, but the value of the expression is promoted to *float*.

One use of the cast is to ensure that the result of a divide is a real number. For example, if we calculated the average of a series of integer test scores without a cast, the result would be an integer. To force a real result, we cast the calculation as shown below.

```
average = (float) totalScores / numScores;
```

In this statement, there is an explicit conversion of totalScores to *float*, and then an implicit conversion of numScores so that it will match. The result of the divide is then a floating-point number to be assigned to average.

But beware! What would be the result of the following expression when a is 3?

```
(float) (a / 10)
```

Are you surprised to find that the result is 0.0? Since no conversions are required to divide integer 3 by integer 10, C simply divides with an integer result, 0. The integer 0 is then explicitly converted to the floating-point 0.0. To get a *float* result, we must cast one of the numbers as shown below.

```
(float) a / 10
```

One final thought about casts: Even when the compiler can correctly cast for you, it is sometimes better to code the cast explicitly as a reminder that the cast is taking place.

Program 3-8 demonstrates the use of explicit casts. In this program, we divide several mixed types. While the results are nonsense, they demonstrate the effect of casting.

PROGRAM 3-8 Explicit Casts

```
/* Demonstrate casting of numeric types.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   char   aChar    = '\0';
   int    intNum1  = 100;
   int    intNum2  =  45;
   double fltNum1  = 100.0;
   double fltNum2  =  45.0;
   double fltNum3;

// Statements
   printf("aChar numeric   :  %3d\n",   aChar);
   printf("intNum1 contains:  %3d\n",   intNum1);
   printf("intNum2 contains:  %3d\n",   intNum2);
   printf("fltNum1 contains:  %6.2f\n", fltNum1);
   printf("fltNum2 contains:  %6.2f\n", fltNum2);

   fltNum3 = (double)(intNum1 / intNum2);
   printf
      ("\n(double)(intNum1 / intNum2): %6.2f\n",
       fltNum3);

   fltNum3 = (double)intNum1 / intNum2;
   printf("(double) intNum1 / intNum2 : %6.2f\n",
          fltNum3);

   aChar = (char)(fltNum1 / fltNum2);
   printf("  (char)(fltNum1 / fltNum2): %3d\n",  aChar);

```

continued

PROGRAM 3-8 Explicit Casts *(continued)*

```
36       return 0;
37   }   // main
```

```
Results:
aChar numeric    :   0
intNum1 contains:  100
intNum2 contains:   45
fltNum1 contains:  100.00
fltNum2 contains:   45.00

(double)(intNum1 / intNum2):    2.00
(double) intNum1 / intNum2 :    2.22
  (char)(fltNum1 / fltNum2):    2
```

Program 3-8 Analysis Study the casts carefully. The only difference between statements 24 and 29 is the use of parentheses around the calculation. In statement 24, both operands are integers so the result of the division is integer, which is then cast to a *double*. In statement 29, `intNum1` is cast to a *double*. The compiler automatically casts `intNum2` to a *double* before the division. The result is therefore a *double*. Finally, in statement 33, we cast the result of the integer division into a character.

3.6 Statements

A statement causes an action to be performed by the program. It translates directly into one or more executable computer instructions.

You may have noticed that we have used a semicolon at the end of the statements in our programs. Most statements need a semicolon at the end; some do not. When we discuss statements, we identify those that do.

Statement Type

C defines eleven types of statements, which are shown in Figure 3-11. In this chapter, we will discuss the first four; the other types will be covered in the future chapters.

Null Statement

The **null statement** is just a semicolon (the terminator) as shown below:

```
;                                      // null statement
```

Although they do not arise often, there are syntactical situations where we must have a statement but no action is required. In these situations, we use the null statement.

Expression Statement

An expression is turned into a statement by placing a semicolon (`;`) after it.

```
expression;                          // expression statement
```

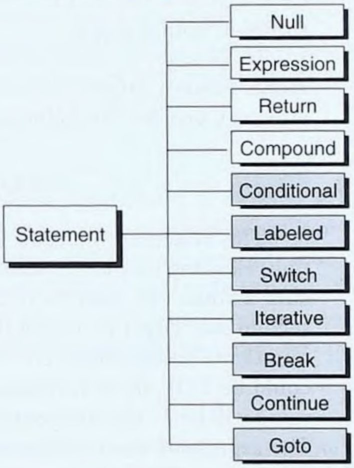


FIGURE 3-11 Types of Statements

When C sees the semicolon, it completes any pending side effects and discards the expression value before continuing with the next statement. An expression without side effects does not cause an action. Its value exists and it can be used, but unless a statement has a side effect, it does nothing.

Let us look at some **expression statements**. First, consider the expression statement

```
a = 2;
```

The effect of the expression statement is to store the value, 2, in the variable a. The value of the expression is 2. After the value has been stored, the expression is terminated (because there is a semicolon), and the value is discarded. C then continues with the next statement.

The next expression statement is a little more complex.

```
a = b = 3;
```

This statement actually has two expressions. If we put parentheses around them, you will be able to see them clearly.

```
a = (b = 3);
```

The parenthesized expression (b = 3) has a side effect of assigning the value 3 to the variable b. The value of this expression is 3. The expression statement now results in the expression value 3 being assigned to the variable a. Since the expression is terminated by the semicolon, its value, 3, is discarded. The effect of the expression statement, therefore, is that 3 has been stored in both a and b.

In Chapter 2 we examined the *scanf* and *printf* functions. These statements present interesting insights into the concepts of expressions and side effects. Consider the following *scanf* function call:

```
ioResult = scanf("%d", &x);
```

This statement has two side effects. The first is found in the *scanf* function. Reading an integer value from the keyboard and placing it into the variable x (note the address operator before the variable) is a side effect. The second side effect is storing the value returned by *scanf*, which represents the number of values that were converted correctly. In this case, the return value could be EOF, 0, or 1. Assuming that the user correctly keys the integer, the value will be 1. The assignment operator stores the return value in ioResult. The expression then terminates, and the *scanf* value is discarded.

In a similar fashion, *printf* has the effect of displaying data on the monitor and returning a value, the number of characters displayed. This is seen in the statement below.

```
numDisplayed = printf
    ("x contains %d, y contains %d\n", x, y);
```

As a general rule, however, the number of characters displayed is discarded without being stored. Therefore, we normally code the above statement as shown below.

```
printf("x contains %d, y contains %d\n", x, y);
```

Now consider the following expression statement. Assume that a has a value of 5 before the expression is evaluated.

```
a++;
```

In this postfix expression, the value of the expression is 5, the value of the variable, a, before it is changed by the side effect. Upon completion of the expression statement, a is incremented to 6. The value of the expression, which is still 5, is discarded since the expression is now complete.

Although they are useless, the following are also expression statements. They are useless because they have no side effect and their values are not assigned to a variable. We usually don't use them, but it is important to know they are syntactically correct expression statements. C will evaluate them, determine their value, and then discard the value.[1]

```
          b;               3;               ;
```

Return Statement

A return statement terminates a function. All functions, including *main*, must have a return statement. When there is no return statement at the end of the function, the system inserts one with a *void* return value.

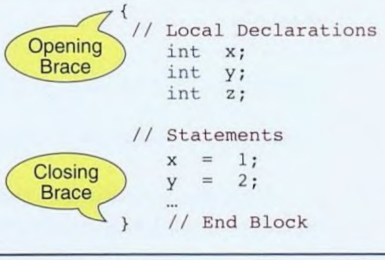
```
return expression;                   // return statement
```

In addition, the return statement can return a value to the calling function. For example, while we have not talked about it or used it, the *scanf* function returns the number of variables successfully read. In the case of *main*, it returns a value to the operating system rather than to another function. In all of our programs, *main* returns 0 to the operating system. A return value of zero tells the operating system that the program executed successfully.

Compound Statements

A **compound statement** is a unit of code consisting of zero or more statements. It is also known as a **block**. The compound statement allows a group of statements to become one single entity. You used a compound statement in your first program when you formed the body of the function *main*. All C functions contain a compound statement known as the function body.

A compound statement consists of an opening brace, an optional declaration and definition section, and an optional statement section, followed by a closing brace. Although both the declaration section and the statement sections are optional, one should be present. If neither is present, then we have a null statement, which doesn't make much sense. Figure 3-12 shows the makeup of a compound statement.



```
{
// Local Declarations
   int  x;
   int  y;
   int  z;

// Statements
   x  =  1;
   y  =  2;
   …
}  // End Block
```

FIGURE 3-12 Compound Statement

[1] In an optimized compiler, the translator determines that there is no effect and generates no code.

One important point to remember is that a compound statement does not need a semicolon. The opening and closing parentheses are simply delimiters for the compound statement. If we put a semicolon after the closing brace, the compiler thinks that we have put an extra null statement after the compound statement. This is poor style, but it does not generate any code or generate a compile error, although it may generate a warning message.

The compound statement does not need a semicolon.

C requires that the declaration section be before any statements[2] within a compound statement block. The code in the declaration section and statement section cannot be intermixed.

The Role of the Semicolon

The semicolon plays an important role in the syntax of the C language. It is used in two different auscultations.

- Every declaration in C is terminated by a semicolon.
- Most statements in C are terminated by a semicolon.

On the other hand, we must be careful not to use a semicolon when it is not needed. A semicolon should not be used with a preprocessor directive such as the include and define. In particular, a semicolon at the end of a define statement can create a problem as discussed in the next section.

Statements and Defined Constants

When we use preprocessor-defined commands, we must be very careful to make sure that we do not create an error. Remember that the defined constant is an automatic substitution. This can cause subtle problems. One common mistake is to place a semicolon at the end of the definition. Since the preprocessor uses a simple text replacement of the name with whatever expression follows, the compiler will generate a compile error if it finds a semicolon at the end of the definition. This problem is seen in the following example:

```
#define SALES_TAX_RATE 0.825;
...
salesTax = SALES_TAX_RATE * salesAmount;
```

After the substitution, the following erroneous code occurs because we coded a semicolon after the constant value:

```
            salesTax = 0.0825; * salesAmount;
```

[2] The one exception to this rule is in the *for* statement, which we study in Chapter 6.

This can be an extremely difficult compile error to figure out because we see the original statement and not the erroneous substitution error. One of the reasons programmers use uppercase for defined constant identifiers is to provide an automatic warning to readers that they are not looking at the real code.

3.7 Sample Programs

This section contains several programs that you should study for programming technique and style.

EXAMPLE 3-4 Calculate Quotient and Remainder

Let's write a program that calculates and prints the quotient and remainder of two integer numbers. The code is shown in Program 3-9.

PROGRAM 3-9 Calculate Quotient and Remainder

```c
/* Calculate and print quotient and remainder of two
   numbers.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int  intNum1;
   int  intNum2;
   int  intCalc;

// Statements
   printf("Enter two integral numbers: ");
   scanf ("%d %d", &intNum1, &intNum2);

   intCalc = intNum1 / intNum2;
   printf("%d / %d is %d", intNum1, intNum2, intCalc);

   intCalc = intNum1 % intNum2;
   printf(" with a remainder of: %d\n", intCalc);

   return 0;
}  // main
```

continued

PROGRAM 3-9 Calculate Quotient and Remainder *(continued)*

```
Results:
Enter two integral numbers: 13 2
13 / 2 is 6 with a remainder of: 1
```

Program 3-9 Analysis Using good programming style, the program begins with documentation about what it does, who created it, and when it was created.

Program 3-9 has no global variable declarations, so after including the standard input/output library, we start immediately with *main*. Following *main* is the opening brace. The matching closing brace is found on line 26.

The most difficult part of this program is figuring out how to get the remainder. Fortunately, C has a modulo operator (%) that does the job for us. The rest of the problem is straightforward.

EXAMPLE 3-5 Print Right Digit

Another problem that requires the use of the modulo operator is to print a digit contained in an integer. Program 3-10 prints the least significant (rightmost) digit of an integer.

PROGRAM 3-10 Print Right Digit of Integer

```
/* Print rightmost digit of an integer.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int  intNum;
   int  oneDigit;

// Statements
   printf("Enter an integral number: ");
   scanf ("%d", &intNum);

   oneDigit = intNum % 10;
   printf("\nThe right digit is: %d", oneDigit);

   return 0;
}  // main
```

```
Results:
```

continued

PROGRAM 3-10 Print Right Digit of Integer *(continued)*

```
Enter an integral number: 185

The right digit is: 5
```

EXAMPLE 3-6 Calculate Average

Program 3-11 reads four integers from the keyboard, calculates their average, and then prints the numbers with their average and the deviation (not the standard deviation, just the difference plus or minus) from the average.

PROGRAM 3-11 Calculate Average of Four Numbers

```c
/* Calculate the average of four integers and print
   the numbers and their deviation from the average.
      Written by:
      Date:
*/
#include <stdio.h>
int main (void)
{
// Local Declarations
   int    num1;
   int    num2;
   int    num3;
   int    num4;
   int    sum;
   float  average;

// Statements
   printf("\nEnter the first number  : ");
   scanf("%d", &num1);
   printf("Enter the second number : ");
   scanf("%d", &num2);
   printf("Enter the third  number : ");
   scanf("%d", &num3);
   printf("Enter the fourth number : ");
   scanf("%d", &num4);

   sum     = num1 + num2 + num3 + num4;
   average = sum / 4.0;

   printf("\n ******** average is %6.2f ******** ",
           average);
   printf("\n");
```

continued

PROGRAM 3-11 Calculate Average of Four Numbers *(continued)*

```

   printf("\nfirst number:  %6d -- deviation: %8.2f",
          num1, num1 - average);
   printf("\nsecond number: %6d -- deviation: %8.2f",
          num2, num2 - average);
   printf("\nthird number:  %6d -- deviation: %8.2f",
          num3, num3 - average);
   printf("\nfourth number: %6d -- deviation: %8.2f",
          num4, num4 - average);

   return 0;
}  // main
```

```
Results:
Enter the first number:  23
Enter the second number: 12
Enter the third  number: 45
Enter the fourth number: 23

******** average is 25.75  ********

first number:      23 -- deviation:    -2.75
second number:     12 -- deviation:   -13.75
third number:      45 -- deviation:    19.25
fourth number:     23 -- deviation:    -2.75
```

Program 3-11 Analysis

Program 3-11 is a little more complex than the previous ones. At the beginning of *main* are several variable declarations, five integers, and a floating-point number. The first four are for the variables read from the keyboard, the fifth is for the sum, and the floating-point number is for the average.

The statements section starts by reading the data. Each read is preceded by a display so the user will know what to do. The specific instructions about what to input are known as **user prompts**. You should always tell the user what input is expected from the keyboard. After the user has keyed the data, the program continues by adding the numbers, placing the total in `sum`, and computing `average`. It then displays the results. Notice that the program displays the results in a format that allows the user to easily verify that the program ran correctly. The program not only prints the average but also repeats each input with its deviation from the average. After completing its work, the program concludes by returning to the operating system.

Look at the results carefully. Note how each series of numbers is aligned so that they can be easily read. Taking the time to align output is one of the things that distinguishes a good programmer from an average programmer. Always pay attention to how your program presents its results to the user. Paying attention to these little details pays off in the long run.

EXAMPLE 3-7

Degrees to Radians

One way to measure an angle in a circle is in degrees. For example, the angle formed by a clock at 3 o'clock is 90°. Another way to measure the angle is in radians. One radian is equal to 57.295779[3] degrees. In Program 3-12 we ask the user to input an angle in radians, and we convert it to degrees.

PROGRAM 3-12 Convert Radians to Degrees

```
/* This program prompts the user to enter an angle
   measured in radians and converts it into degrees.
      Written by:
      Date:
*/
#include <stdio.h>

#define DEGREE_FACTOR 57.295779

int main (void)
{
// Local Declarations
   double  radians;
   double  degrees;

// Statements
   printf("Enter the angle in radians: ");
   scanf("%lf", &radians);

   degrees = radians * DEGREE_FACTOR;

   printf("%6.3f radians is %6.3f degrees\n",
            radians, degrees);
   return 0;
}  // main
```

```
Results:
Enter the angle in radians: 1.57080
 1.571 radians is 90.000 degrees
```

Program 3-12 Analysis In this short program we introduce the defined constant. Defined constants are an excellent way to document factors in a program. We could have just as easily used a memory constant. Factors are usually placed at the beginning of the program where they can easily be found and changed as necessary. Remember that the area before *main* is global and that it should be used only for declarations and constants—not variables.

[3] More precisely, a radian is (180 / π).

EXAMPLE 3-8 Calculate Sales Total

Program 3-13 calculates a sale given the unit price, quantity, discount rate, and sales tax rate.

PROGRAM 3-13 Calculate Sales Total

```
/* Calculates the total sale given the unit price,
   quantity, discount, and tax rate.
      Written by:
      Date:
*/
#include <stdio.h>

#define TAX_RATE 8.50

int main (void)
{
// Local Declarations
   int    quantity;

   float  discountRate;
   float  discountAm;
   float  unitPrice;
   float  subTotal;
   float  subTaxable;
   float  taxAm;
   float  total;

// Statements
   printf("\nEnter number of items sold:       ");
   scanf("%d", &quantity);

   printf("Enter the unit price:               ");
   scanf("%f", &unitPrice);

   printf("Enter the discount rate (per cent): ");
   scanf("%f", &discountRate);

   subTotal    = quantity * unitPrice;
   discountAm  = subTotal * discountRate / 100.0;
   subTaxable  = subTotal - discountAm;
   taxAm = subTaxable * TAX_RATE/ 100.00;
   total = subTaxable + taxAm;

   printf("\nQuantity sold:      %6d\n", quantity);
```

continued

PROGRAM 3-13 Calculate Sales Total *(continued)*

```
   printf("Unit Price of items: %9.2f\n", unitPrice);
   printf("                     ------------\n");

   printf("Subtotal :            %9.2f\n", subTotal);
   printf("Discount:            -%9.2f\n", discountAm);
   printf("Discounted total:     %9.2f\n", subTaxable);
   printf("Sales tax:           +%9.2f\n", taxAm);
   printf("Total sale:           %9.2f\n", total);

   return 0;
}  // main
```

```
Results:
Enter number of items sold:          34
Enter the unit price:                12.89
Enter the discount rate (per cent): 7

Quantity sold:            34
Unit Price of items:      12.89
                      ------------
Subtotal :               438.26
Discount:            -    30.68
Discounted total:        407.58
Sales tax:           +    34.64
Total sale:              442.23
```

Program 3-13 Analysis Look at the results of this program carefully. Do you see any problems? Just because a program runs doesn't mean that it is running correctly. In this case, the total is incorrect (407.58 + 34.64 is not equal to 442.23). The problem is created by the floating-point arithmetic and rounding errors. If we wanted absolute accuracy, we would have to do the arithmetic in integer (cents) and then divide by 100 to print the report.

EXAMPLE 3-9 Calculate Student Score

Program 3-14 calculates the average score for a student. The class has four quizzes (30%), two midterms (40%), and a final (30%). The maximum score for all quizzes and exams is 100 points.

PROGRAM 3-14 Calculate Student Score

```
/* Calculate a student's average score for a course
   with 4 quizzes, 2 midterms, and a final. The quizzes
   are weighted 30%, the midterms 40%, & the final 30%.
      Written by:
```

continued

PROGRAM 3-14 Calculate Student Score *(continued)*

```
      Date:
*/
#include <stdio.h>

#define QUIZ_WEIGHT        30
#define MIDTERM_WEIGHT     40
#define FINAL_WEIGHT       30
#define QUIZ_MAX          400.00
#define MIDTERM_MAX       200.00
#define FINAL_MAX         100.00

int main (void)
{
// Local Declarations
   int    quiz1;
   int    quiz2;
   int    quiz3;
   int    quiz4;
   int    totalQuiz;
   int    midterm1;
   int    midterm2;
   int    totalMidterm;
   int    final;

   float  quizPercent;
   float  midtermPercent;
   float  finalPercent;
   float  totalPercent;

// Statements
   printf("=========== QUIZZES ==================\n");
   printf("Enter the score for the first quiz:  ");
   scanf("%d", &quiz1);
   printf("Enter the score for the second quiz: ");
   scanf("%d", &quiz2);
   printf("Enter the score for the third quiz:  ");
   scanf("%d", &quiz3);
   printf("Enter the score for the fourth quiz: ");
   scanf("%d", &quiz4);

   printf("============ MIDTERM  =============\n");
   printf("Enter the score for the first midterm:  ");
   scanf("%d", &midterm1);
   printf("Enter the score for the second midterm: ");
```

continued

PROGRAM 3-14 Calculate Student Score *(continued)*

```
  scanf("%d", &midterm2);

  printf("============== FINAL ==============\n");
  printf("Enter the score for the final: ");
  scanf("%d", &final);
  printf("\n");

  totalQuiz = quiz1 + quiz2 + quiz3 + quiz4;
  totalMidterm = midterm1 + midterm2;

  quizPercent =
     (float)totalQuiz * QUIZ_WEIGHT / QUIZ_MAX;
  midtermPercent =
     (float)totalMidterm * MIDTERM_WEIGHT / MIDTERM_MAX;
  finalPercent =
     (float)final * FINAL_WEIGHT / FINAL_MAX;

  totalPercent =
        quizPercent + midtermPercent + finalPercent;

  printf("First Quiz  %4d\n",   quiz1);
  printf("Second Quiz %4d\n",   quiz2);
  printf("Third Quiz  %4d\n",   quiz3);
  printf("Fourth Quiz %4d\n",   quiz4);
  printf("Quiz Total  %4d\n\n", totalQuiz);

  printf("First Midterm  %4d\n",   midterm1);
  printf("Second Midterm %4d\n",   midterm2);
  printf("Total Midterms %4d\n\n", totalMidterm);

  printf("Final          %4d\n\n", final);

  printf("Quiz    %6.1f%%\n" , quizPercent);
  printf("Midterm %6.1f%%\n" , midtermPercent);
  printf("Final   %6.1f%%\n" , finalPercent);
  printf("--------------\n");
  printf("Total   %6.1f%%\n" , totalPercent);

  return 0;
} // main
```

```
Results:
=========== QUIZZES ================
Enter the score for the first quiz:  98
```

continued

PROGRAM 3-14 Calculate Student Score *(continued)*

```
Enter the score for the second quiz: 89
Enter the score for the third quiz:  78
Enter the score for the fourth quiz: 79
============ MIDTERM  ============
Enter the score for the first midterm:  90
Enter the score for the second midterm: 100
============== FINAL =============
Enter the score for the final: 92

First Quiz      98
Second Quiz     89
Third Quiz      78
Fourth Quiz     79
Quiz Total     344

First Midterm      90
Second Midterm    100
Total Midterms    190

Final                92

Quiz        25.8%
Midterm     38.0%
Final       27.6%
--------------
Total       91.4%
```

Program 3-14 Analysis This rather long program contains several points to consider. First, note how the program starts with a series of defined constants. Putting the definitions of constant values at the beginning of the program does two things: (1) It gives them names that we can use in the program, and (2) it makes them easy to change.

Now study the statements. Notice how they are grouped? By putting a blank line between a group of related statements, we separate them, much as we would separate paragraphs in a report. This makes it easy for the user to follow the program.

Finally, study the input and output. Notice that the user was prompted for all input with clear instructions. We even divided the input with headings. The output is also divided, making it easy to read. It would be even easier to read if we aligned all the amounts, but the techniques for doing so will not be introduced until Chapter 7.

3.8 Software Engineering

In this section we discuss three concepts that, although technically not engineering principles, are important to writing clear and understandable programs.

KISS

Keep It Simple and Short (**KISS**[4]) is an old programming principle. Unfortunately, many programmers tend to forget it, especially the simple part. They seem to feel that just because they are working on a complex problem, the solution must be complex, too. That is simply not true. Good programmers solve the problem in the simplest possible way; they do not contribute to a complex situation by writing obscure and complex code.

A trivial example will make the point. If you were writing a program that reads floating-point numbers from the keyboard, you would not program it so that the user had to enter the integral portion of the number first and then the fractional part. Although this would work, it is unnecessarily complex, even though it might be a fun way to solve the problem.

Unfortunately, C provides many operators and expression rules that make it easy for a programmer to write obscure and difficult to follow code. Your job as a programmer is to make sure that your code is always easy to read. Your code should be unambiguous: It should not be written so that it is easy to misread it.

Another old structured programming principle is that a function should not be larger than one page of code. Now, for online programming in a workstation environment, a function should be no longer than one screen—about 20 lines of code. By breaking a problem down into small, easily understood parts, we simplify it. Then we reassemble the simple components into a simple solution to a complex problem.

Blocks of code should be no longer than one screen.

One element of the C language that tends to complicate programs, especially for new programmers, is side effects. We explained in Section 3.4, "Evaluating Expressions," that side effects can lead to confusing and different results depending on the code. You need to fully understand the effects when you write C code. If you are unsure of the effects, then simplify your logic until you are sure.

Parentheses

One programming technique to simplify code is to use parentheses, even when unnecessary. While this may lead to a few extra keystrokes, it can save hours of debugging time created by a misunderstanding of the precedence

[4] KISS originally had a different, rather insulting, meaning. We prefer this interpretation.

and associativity rules. Whenever a statement contains multiple expressions, use parentheses to ensure that the compiler will interpret it as you intended.

> **Computers do what you tell them to do, not what you *intended* to tell them to do. Make sure your code is as clear and simple as possible.**

User Communication

You should always make sure you communicate with your user from the very first statement in your program to the very last. As mentioned previously, we recommend that you start your program with a message that identifies the program and end with a display that says the program is done.

When you give your user instructions, make sure that they are clear and understandable. In Program 3-13, we used three statements to give the user complete and detailed instructions on what we wanted entered. We could have simply said

```
"Enter data"
```

but that would have been vague and subject to interpretation. How would the user know what specific data were required? For each input in Program 3-13, we told the users exactly what data we needed in terms that they understand. If you don't tell users exactly what data to input, they may do anything they feel like, which is usually not what you wanted or expected.

One common mistake made by new programmers is to forget to tell the user anything. What do you think would be the user's response to Program 3-15 when confronted with a blank screen and a computer that is doing nothing?

PROGRAM 3-15 Program That Will Confuse the User

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

   scanf("%d%d", &i, &j);
   sum = i + j;
   printf("The sum of %d & %d is %d\n", i, j, sum);
   return 0;
} //  main
```

We now rewrite the program with clear user communication. With the addition of one print statement, the user knows exactly what is to be done.

PROGRAM 3-16 Program That Will Not Confuse the User

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

   printf("Enter two integers and key <return>\n");
   scanf("%d%d", &i, &j);
   sum = i + j;
   printf("The sum of %d & %d is %d\n", i, j, sum);
   return 0;
}  //  main
```

```
Results:
Enter two integers and key <return>
4 5
The sum of 4 & 5 is 9
```

We will return to these three concepts from time to time when we introduce new structures that tend to be confusing or misunderstood.

3.9 Tips and Common Errors

1. Be aware of expression side effects. They are one of the main sources of confusion and logical errors in a program.
2. Use decrement/increment operators wisely. Understand the difference between postfix and prefix decrement/increment operators before using them.
3. Add parentheses whenever you feel they will help to make an expression clearer.
4. It is a compile error to use a variable that has not been defined.
5. It is a compile error to forget the semicolon at the end of an expression statement.
6. It is a compile error to code a variable declaration or definition once you have started the statement section. To help yourself remember this rule, use comments to separate these two sections within a function.
7. It is most likely a compile error to terminate a defined constant (*#define*) with a semicolon. This is an especially difficult error to decipher because you will not see it in your code—you see the code you wrote, not the code that the preprocessor substituted.
8. It is a compile error when the operand on the left of the assignment operator is not a variable. For example, `a + 3` is not a variable and cannot receive the value of `b * c`.

```
(a + 3) = b * c;
```

9. It is a compile error to use the increment or decrement operators with any expression other than a variable identifier. For example, the following code is an error:

```
(a + 3)++
```

10. It is a compile error to use the modulus operator (`%`) with anything other than integers.
11. It is a logic error to use a variable before it has been assigned a value.
12. It is a logic error to modify a variable in an expression when the variable appears more than once.

3.10 Key Terms

additive expression
assignment expression
associativity
left-to-right associativity
multiplicative expression
name

binary expression
block
cast
complex expression
compound statement
conversion rank
demotion
explicit type conversion
expression
expression statement
implicit type conversion
KISS
operand
operator
postfix expression
precedence
primary expression
promotion
right-to-left associativity
side effect
simple expression
unary expression
user prompts

3.11 Summary

- An expression is a sequence of operators and operands that reduces to a single value.
- An operator is a language-specific token that requires an action to be taken.
- An operand is the recipient of the action.
- C has six kinds of expressions: primary, postfix, prefix, unary, binary, and ternary.
- The most elementary type of expression is a primary expression. A primary expression is an expression made up of only one operand. It can be a name, a constant, or a parenthesized expression.
- A postfix expression is an expression made up of an operand followed by an operator. You studied function call and postfix increment/decrement expressions in this chapter.
- A unary expression is an expression made up of an operator followed by an operand. You studied six in this chapter: prefix increment/decrement, *sizeof*, plus/minus, and cast expressions.
- A binary expression is an expression made up of two operands with an operator between them. You studied multiplicative, additive, and assignment expressions in this chapter.
- Precedence is a concept that determines the order in which different operators in a complex expression act on their operands.
- Associativity defines the order of evaluation when operators have the same precedence.
- The side effect of an expression is one of the unique phenomena in C. An expression can have a side effect in addition to a value.

❑ To evaluate an expression, we must follow the rules of precedence and associativity.

❑ A statement causes an action to be performed by the program.

❑ Although C has eleven different types of statements, you studied only four types in this chapter:

1. A null statement is just a semicolon.
2. An expression statement is an expression converted to a statement by keeping the side effect and discarding the value.
3. A return statement terminates a function.
4. A compound statement is a combination of statements enclosed in two braces.

❑ KISS means "Keep It Simple and Short."

❑ One of the important recommendations in software engineering is the use of parentheses when they can help clarify your code.

❑ Another recommendation in software engineering is to communicate clearly with the user.

3.12 Practice Sets

Review Questions

1. A unary expression consists of only one operand with no operator.
   a. True
   b. False
2. The left operand in an assignment expression must be a single variable.
   a. True
   b. False
3. Associativity is used to determine which of several different expressions is evaluated first.
   a. True
   b. False
4. Side effect is an action that results from the evaluation of an expression.
   a. True
   b. False
5. An expression statement is terminated with a period.
   a. True
   b. False

6. A(n) ______________ is a sequence of operands and operators that reduces to a single value.

   a. expression
   b. category
   c. formula
   d. function
   e. value

7. Which of the following is a unary expression?

   a. `i + j`
   b. `+a`
   c. `c++`
   d. `scanf(…)`
   e. `x *= 5`

8. The ______________ expression evaluates the operand on the right side of the operator and places its value in the variable on the left side of the operator.

   a. additive
   b. assignment
   c. multiplicative
   d. postfix
   e. primary

9. ______________ is used to determine the order in which different operators in a complex expression are evaluated.

   a. associativity
   b. evaluation
   c. category
   d. precedence
   e. side effect

10. ______________ is an action that results from the evaluation of an expression.

    a. associativity
    b. evaluation
    c. category
    d. precedence
    e. side effect

11. Which of the following statements about mixed expressions is false?

    a. A cast cannot be used to change an assigned value.
    b. An explicit cast can be used to change the expression type.
    c. An explicit cast on a variable changes its type in memory.
    d. An implicit cast is generated by the compiler automatically when necessary.
    e. Constant casting is done by the compiler automatically.

12. Which of the following statements about compound statements is false?

- **a.** A compound statement is also known as a block.
- **b.** A compound statement is enclosed in a set of braces.
- **c.** A compound statement must be terminated by a semicolon.
- **d.** The declaration and definition section in a compound statement is optional.
- **e.** The statement section in a compound statement is optional.

Exercises

13. Which of the following expressions are not postfix expressions?

- **a.** `x++`
- **b.** `--x`
- **c.** `scanf (…)`
- **d.** `x * y`
- **e.** `++x`

14. Which of the following are not unary expressions?

- **a.** `++x`
- **b.** `--x`
- **c.** `sizeof (x)`
- **d.** `+5`
- **e.** `x = 4`

15. Which of the following is not a binary expression?

- **a.** `3 * 5`
- **b.** `x += 6`
- **c.** `y = 5 + 2`
- **d.** `z - 2`
- **e.** `y % z`

16. Which of the following is not a valid assignment expression?

- **a.** `x = 23`
- **b.** `4 = x`
- **c.** `y % = 5`
- **d.** `x = 8 = 3`
- **e.** `x = r = 5`

17. If originally `x = 4`, what is the value of `x` after the evaluation of the following expression?

- **a.** `x = 2`
- **b.** `x += 4`
- **c.** `x += x +3`
- **d.** `x *= 2`
- **e.** `x /= x +2`

18. If originally `x = 3` and `y = 5`, what is the value of x and y after each of the following expressions?

a. `x++ + y`
b. `++x`
c. `x++ + y++`
d. `++x + 2`
e. `x-- - y--`

19. What is the value of each of the following expressions?

a. `24 - 6 * 2`
b. `-15 * 2 + 3`
c. `72 / 5`
d. `72 % 5`
e. `5 * 2 / 6 + 15 % 4`

20. What is the value of each of the following expressions?

a. `6.2 + 5.1 * 3.2`
b. `2.0 + 3.0 / 1.2`
c. `4.0 * (3.0 + 2.0 / 6.0)`
d. `6.0 / (2.0 + 4.0 * 1.2)`
e. `2.7 + 3.2 - 5.3 * 1.1`

21. Given the following definitions, which of the following statements are valid assignments?

```
#define NUM10 10
int x;
int y = 15;
```

a. `x = 5;`
b. `y = 5;`
c. `x = y = 50;`
d. `x = 50 = y;`
e. `x = x + 1;`
f. `y = 1 + NUM10;`
g. `5 = y;`

22. If originally `x = 2`, `y = 3`, and `z = 2`, what is the value of each of the following expressions?

a. `x++ + y++`
b. `++x - --z`
c. `--x + y++`
d. `x-- + x-- - y--`
e. `x + y-- - x + x++ - --y`

23. If originally `x = 2`, `y = 3`, and `z = 1`, what is the value of each of the following expressions?

a. `x + 2 / 6 + y`
b. `y - 3 * z + 2`
c. `z - (x + z) % 2 + 4`

d. `x - 2 * (3 + z) + y`

e. `y++ + z-- + x++`

24. If `x = 2945`, what is the value of each of the following expressions?

a. `x % 10`

b. `x / 10`

c. `(x / 10) % 10`

d. `x / 100`

e. `(x / 100) % 10`

25. What is the output from the following code fragment?

```
int a;
int b;
a = b = 50;
printf ("%4d %4d", a, b);
a = a * 2;
b = b / 2;
printf ("%4d %4d", a, b);
```

Problems

26. Given the following pseudocode, write a program that executes it. Use floating-point types for all values.

```
Algorithm Problem26
  read x
  read y
  compute p = x * y
  compute s = x + y
  total = $s^2$ + p * (s - x) * (p + y)
  print total
End Problem26
```

27. Write a program that reads two integers from the keyboard, multiplies them, and then prints the two numbers and their product.

28. Write a program that extracts and prints the rightmost digit of the integral portion of a *float*.

29. Write a program that extracts and prints the second rightmost digit of the integral portion of a *float*.

30. Write a program that calculates the area and perimeter of a rectangle from a user-supplied (*scanf*) length and width.

31. We are all familiar with the fact that angles are measured in degrees, minutes, and seconds. Another measure of an angle is a radian. A radian is the angle formed by two radii forming an arc that is equal to the radius of their circle. One radian equals 57.295779 degrees. Write a program

that converts degrees into radians. Provide good user prompts. Include the following test data in your run:

```
90° is 1.57080 radians
```

32. The formula for converting centigrade temperatures to Fahrenheit is:

$$F = 32 + \left(C \times \left(\frac{180.0}{100.0}\right)\right)$$

Write a program that asks the user to enter a temperature reading in centigrade and then prints the equivalent Fahrenheit value. Be sure to include at least one negative centigrade number in your test cases.

33. Write a program that changes a temperature reading from Fahrenheit to Celsius using the following formula:

```
Celsius = (100.0 / 180.0) * (Fahrenheit - 32)
```

Your program should prompt the user to enter a Fahrenheit temperature. It then calculates the equivalent Celsius temperature and displays the results as shown below.

```
Enter the temperature in Fahrenheit: 98.6
Fahrenheit temperature is:  98.6
Celsius temperature is:     37.0
```

34. Write the C code for each of the following formulas. Assume that all variables are defined as *double*.

a.

$$\mathbf{KinEn} = \frac{mv^2}{2}$$

b.

$$\mathbf{res} = \frac{b + c}{2bc}$$

35. Write the C code to calculate and print the next two numbers in each of the following series. You may use only one variable in each problem.

a. 0, 5, 10, 15, 20, 25, ?, ?

b. 0, 2, 4, 6, 8, 10, ?, ?

c. 1, 2, 4, 8, 16, 32, ?, ?

Projects

36. Write a program that converts and prints a user-supplied measurement in inches into
    - **a.** foot (12 inches)
    - **b.** yard (36 inches)
    - **c.** centimeter (2.54/inch)
    - **d.** meter (39.37 inches)

37. A Fibonacci number is a member of a set in which each number is the sum of the previous two numbers. (The Fibonacci series describes a form of a spiral.) The series begins

```
0, 1, 1, 2, 3, 5, 8, 13, 21, …
```

Write a program that calculates and prints the next three numbers in the Fibonacci series. You are to use only three variables, `fib1`, `fib2`, and `fib3`.

38. Write a program that prompts a user for an integer value in the range 0 to 32,767 and then prints the individual digits of the numbers on a line with three spaces between the digits. The first line is to start with the leftmost digit and print all five digits; the second line is to start with the second digit from the left and print four digits, and so forth. For example, if the user enters 1234, your program should print

```
0   1   2   3   4
1   2   3   4
2   3   4
3   4
4
```

39. Write a program to create a customer's bill for a company. The company sells only five different products: TV, VCR, Remote Controller, CD Player, and Tape Recorder. The unit prices are $400.00, $220, $35.20, $300.00, and $150.00, respectively. The program must read the quantity of each piece of equipment purchased from the keyboard. It then calculates the cost of each item, the subtotal, and the total cost after an 8.25% sales tax.

The input data consist of a set of integers representing the quantities of each item sold. These integers must be input into the program in a user-friendly way; that is, the program must prompt the user for each quantity as shown below. The numbers in boldface show the user's answers.

```
How Many TVs Were Sold? 3
How Many VCRs Were Sold? 5
How Many Remote Controllers Were Sold? 1
How Many CDs Were Sold? 2
How Many Tape Recorders Were Sold? 4
```

The format for the output is shown in Figure 3-13.

```
QTY     DESCRIPTION       UNIT PRICE     TOTAL PRICE
---     -----------       ----------     -----------
XX      TV                    400.00         XXXX.XX
XX      VCR                   220.00         XXXX.XX
XX      REMOTE CTRLR           35.20         XXXX.XX
XX      CD PLAYER             300.00         XXXX.XX
XX      TAPE RECORDER         150.00         XXXX.XX
                                           ---------
                            SUBTOTAL        XXXXX.XX
                            TAX              XXXX.XX
                            TOTAL           XXXXX.XX
```

FIGURE 3-13 Output Format for Project 39

Use either defined constants or memory constants for the unit prices and the tax rate. Use integer variables to store the quantities for each item. Use floating-point variables to store the total price for each item, the bill subtotal, the tax amount, and the total amount of the bill. Run your program twice with the following data:

```
SET 1 → 2  1  4  1  2
SET 2 → 3  0  2  0 21
```

Chapter 4 Functions

In Chapter 2, we introduced data types. In the preceding chapters, we used only the three non-derived types, *void*, integral, and floating-point. Although these types are very useful, they can solve only a limited number of problems. There are six derived types in C, as shown in Figure 4-1.

In this chapter we introduce the first derived type, the function type. The function type is derived from its return type. The return type can be any type except an array, which we introduce in Chapter 8, or a function type.

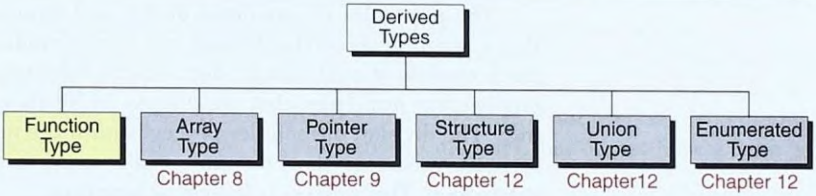


FIGURE 4-1 Derived Types

Objectives

❑ To design and implement programs with more than one function

❑ To be able to design multi-function programs using structure charts

❑ To understand the purpose of the function declaration, call, and definition

❑ To understand the four basic function designs

❑ To understand how two functions communicate through parameters

❑ To understand the differences between global and local scope

❑ To understand the software engineering principles of functional cohesion and top–down development

4.1 Designing Structured Programs

The programs we have presented so far have been very simple. They solved problems that could be understood without too much effort. As we consider larger and larger programs, however, you will discover that it is not possible to understand all aspects of such programs without somehow reducing them to more elementary parts.

Breaking a complex problem into smaller parts is a common practice. For example, suppose that for your vacation this year you decide to drive in a circular route that will allow you to visit as many national parks as possible in two weeks. Your requirements for this problem are very simple: Visit as many parks as possible in two weeks. But how will you do it? You might first gather some data about national parks and then calculate the distance between each of them to figure out the travel time. Next, you would estimate how much time it would take to visit each park. Finally, you would put all your data together to plan your itinerary. Then you would make your motel and camp reservations and any other arrangements that had to be in place in advance.

The planning for large programs is similar. First, you must understand the problem as a whole; then you must break it into simpler, understandable parts. We call each of these parts of a program a *module* and the process of subdividing a problem into manageable parts **top–down design**.

The principles of top–down design and structured programming dictate that a program should be divided into a main module and its related modules. Each module should also be divided into submodules according to software engineering principles that we discuss in Section 4.8, "Software Engineering." The division of modules proceeds until the module consists only of elementary processes that are intrinsically understood and cannot be further subdivided. This process is known as *factoring*.

In top–down design, a program is divided into a main module and its related modules. Each module is in turn divided into submodules until the resulting modules are intrinsic; that is, until they are implicitly understood without further division.

Top–down design is usually done using a visual representation of the modules known as a structure chart. The structure chart shows the relation between each module and its submodules. The rules for reading and creating structure charts are also covered in "Software Engineering," but at this point, we only need a few simple rules. First, the structure chart is read top–down, left–right. Referring to Figure 4-2, first we read Main Module. In this case, Main Module represents our entire set of code to solve the problem.

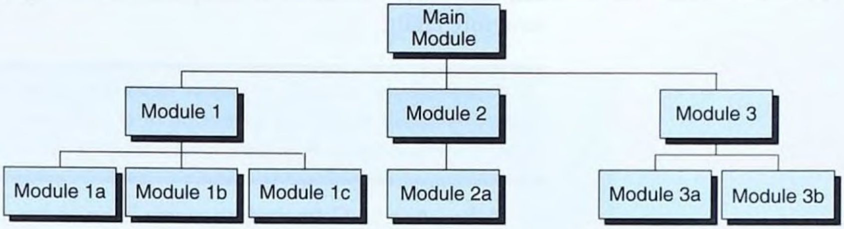


FIGURE 4-2 Structure Chart

Moving down and left, we then read Module 1. On the same level with Module 1 are Module 2 and Module 3. The main module consists of three submodules. At this point, however, we are dealing only with Module 1. We now note that Module 1 is further subdivided into three modules, Module 1a, Module 1b, and Module 1c. To write the code for Module 1, therefore, we will need to write code for its three submodules. What does this concept say about writing the code for the Main Module?

Now for some more terminology. The Main Module is known as a *calling* module because it has submodules. Each of the submodules is known as a *called* module. But, because Modules 1, 2, and 3 also have submodules, they are also calling modules; they are both called and calling modules.

Communication between modules in a structure chart is allowed only through a calling module. If Module 1 needs to send data to Module 2, the data must be passed through the calling module, Main Module. No communication can take place directly between modules that do not have a calling–called relationship.

With this understanding, how can Module 1a send data to Module 3b? It first sends the data to Module 1, which in turn sends it to the Main Module, which passes it to Module 3, and then on to Module 3b. Although this sounds complex, it is easily done.

The technique used to pass data to a function is known as **parameter passing.** The parameters are contained in a list that is a definition of the data passed to the function by the caller. The list serves as the formal declaration of the data types and names.

4.2 Functions in C

In C, the idea of top–down design is done using functions. A C program is made of one or more functions, one and only one of which must be named *main*. The execution of the program always starts and ends with *main*, but it can call other functions to do special tasks. Figure 4-3 shows a C program structure chart.

In C, a program is made of one or more functions, one and only one of which must be called main. The execution of the program always starts with main, but it can call other functions to do some part of the job.

A function in C (including *main*) is an independent module that will be called to do a specific task. A **called function** receives control from a **calling function**. When the called function completes its task, it returns control to the calling function. It may or may not return a value to the caller. The function *main* is called by the operating system; *main* in turn calls other functions. When *main* is complete, control returns to the operating system.

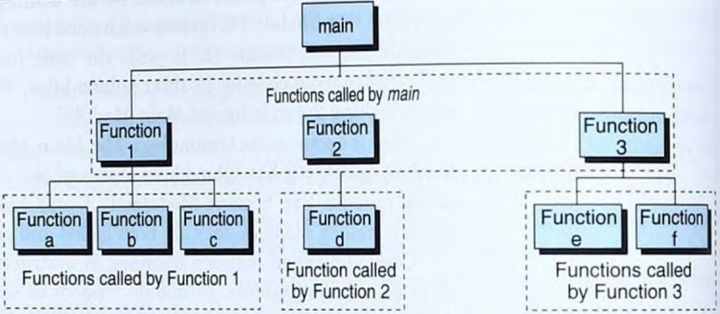


FIGURE 4-3 Structure Chart for a C Program

In general, the purpose of a function is to receive zero or more pieces of data, operate on them, and return at most one piece of data.[1] At the same time, a function can have a side effect. A function side effect is an action that results in a change in the state of the program. If a side effect occurs, it occurs while the function is executing and before the function returns. The side effect can involve accepting data from outside the program, sending data out of the program to the monitor or a file, or changing the value of a variable in the calling function. The function concept is shown in Figure 4-4.

[1] In Chapter 12 we discuss how to return multiple data values using a structure.

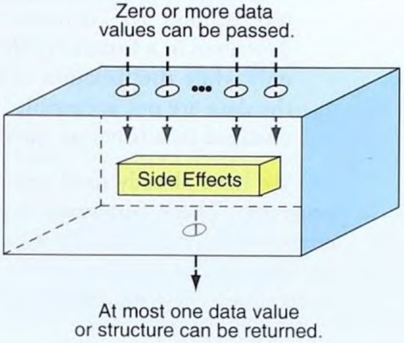

Zero or more data values can be passed.
Side Effects
At most one data value or structure can be returned.

FIGURE 4-4 Function Concept

Several advantages are associated with the use of functions in C or any other language. The major advantages are:

1. As already described, problems can be factored into understandable and manageable steps.
2. Functions provide a way to reuse code that is required in more than one place in a program. Assume, for instance, that a program requires that we compute the average of a series of numbers in five different parts of the program. Each time the data are different. We could write the code to compute the average five times, but this would take a lot of effort. Also, if we needed to change the calculation, we would have to find all five places that use it to change each of them. It is much easier to write the code once as a function and then call it whenever we need to compute the average.

A function in C can have a return value, a side effect, or both. The side effect occurs before the value is returned. The function's value is the value in the expression of the return statement. A function can be called for its value, its side effect, or both.

3. This advantage to using functions is closely tied to reusing code. Like many languages, C comes with a rich and valuable library. For example, there is a math library, *math.h*, that contains almost any mathematical or statistical function that we will ever need. These C libraries provide standard functions that make our work as a programmer much easier. We can also create personal and project libraries that make developing systems easier. Appendix F, "Function Libraries," documents many of the functions included with the C language.

4. We use functions to protect data. This is a rather complex idea that centers around the concept of *local* data. Local data consist of data described in a function. These data are available only to the function and only while the function is executing. When the function is not running, the data are not accessible. Data in one function, then, cannot be seen or changed by a function outside of its scope.

We have already used several functions, such as *scanf* and *printf*, in our programs. These functions, however, were written by the creator of the C environment, not by us. Now we will start writing our own functions. Before we discuss the details, however, let's look at Program 4-1, which demonstrates how we write and call a function.

PROGRAM 4-1 Sample Program with Subfunction

```
/* This program demonstrates function calls by calling
   a small function to multiply two numbers.
      Written by:
      Date:
*/
#include <stdio.h>

// Function Declarations
int multiply (int num1, int num2);

int main (void)
{
// Local Declarations
   int multiplier;
   int multiplicand;
   int product;

// Statements
   printf("Enter two integers: ");
   scanf ("%d%d", &multiplier, &multiplicand);

   product = multiply (multiplier, multiplicand);

   printf("Product of %d & %d is %d\n",
            multiplier, multiplicand, product);
   return 0;
} // main

/* ================= multiply ==================
   Multiples two numbers and returns product.
      Pre   num1 & num2 are values to be multiplied
      Post  product returned
```

continued

PROGRAM 4-1 Sample Program with Subfunction *(continued)*

```
*/
int multiply (int num1, int num2)
{
// Statements
   return (num1 * num2);
} // multiply
```

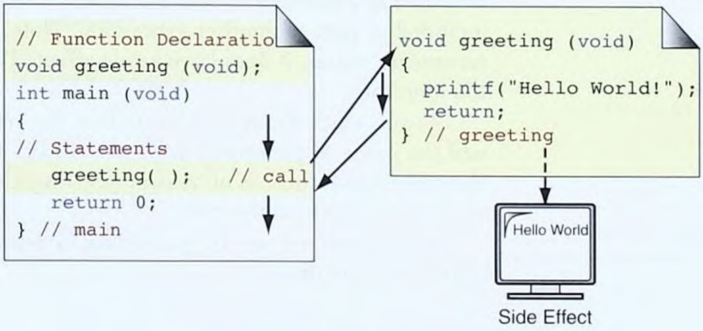
```
Results:
Enter two integers: 17 21
Product of 17 & 21 is 357
```

4.3 User-Defined Functions

Like every other object in C, functions must be both declared and defined. The function declaration, which needs to be done before the function call, gives the whole picture of the function that needs to be defined later. The declaration mentions the name of the function, the return type, and the type and order of formal parameters. In other words, the declaration uses only the header of the function definition ended in a semicolon. The **function definition**, which is traditionally coded after the function that makes the call, contains the code needed to complete the task.

> **A function name is used three times: for declaration, in a call, and for definition.**

Figure 4-5 shows the interrelationships among these function components. As you study Figure 4-5, note that the function name is used three times: when the function is declared, when it is called, and when it is defined.



```
// Function Declaratio
void greeting (void);
int main (void)
{
// Statements
   greeting( );     // call
   return 0;
} // main
```

```
void greeting (void)
{
  printf("Hello World!");
  return;
} // greeting
```

FIGURE 4-5 Declaring, Calling, and Defining Functions

Basic Function Designs

We classify the basic function designs by their return values and their parameter lists. Functions either return a value or they don't. Functions that don't return a value are known as *void* functions. Some functions have parameters and some don't. Combining return types and parameter lists results in four basic designs: *void* functions with no parameters, *void* functions with parameters, functions that return a value but have no parameters, and functions that return a value and have parameters. We discuss these four designs in the sections that follow.

void Functions without Parameters

A *void* function can be written with no parameters. The `greeting` function in Figure 4-5 receives nothing and returns nothing. It has only a side effect, displaying the message, and is called only for that side effect.

The call still requires parentheses, however, even when no parameters are present. When we make a call to a function with no parameters, it is tempting to leave the parentheses off the call. Although this is valid syntax, it is not what we intended. Without the parentheses, it is not a function call.

Because a *void* function does not have a value, it can be used only as a statement; it cannot be used in an expression. Examine the call to the `greeting` function in Figure 4-5. This call stands alone as a statement. Including this call in an expression, as shown below, would be an error.

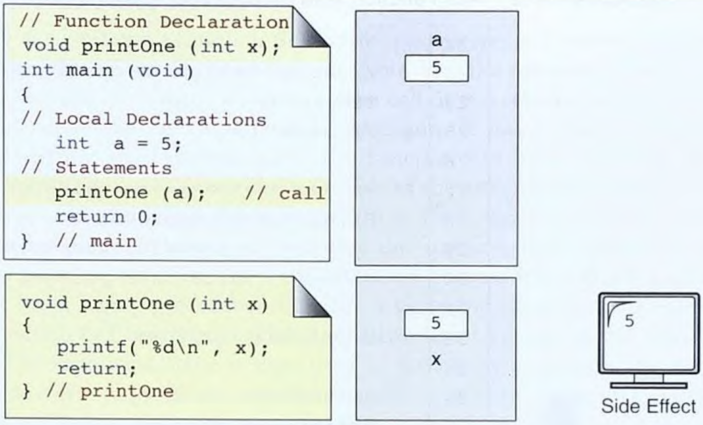
```
result = greeting();            // Error. Void function
```

void Functions with Parameters

Now let's call a function that has parameters but still returns *void*. The function `printOne`, as seen in Figure 4-6, receives an integer parameter. Since this function returns nothing to the calling function, *main*, its return type is *void*. As with the `greeting` function discussed previously, this function must be coded as a stand-alone call because it does not return a value; it cannot be included as part of another expression. Note, however, that while `printOne` returns no values, it does have a side effect: The parameter value is printed to the monitor.

As you study Figure 4-6, note that the name of the variable in *main* (`a`) and the name of parameter in `printOne` (`x`) do not have to be the same. On the other hand, there is no reason why they can't be the same if that makes it easier to understand the code.

In Program 4-2 we use `printOne` to demonstrate that a function can be called multiple times.



```
// Function Declaration
void printOne (int x);
int main (void)
{
// Local Declarations
   int  a = 5;
// Statements
   printOne (a);    // call
   return 0;
}  // main
```

```
void printOne (int x)
{
   printf("%d\n", x);
   return;
} // printOne
```

FIGURE 4-6 *void* Function with Parameters

PROGRAM 4-2 *void* Function with a Parameter

```
/* This program demonstrates that one function can be
   called multiple times.
      Written by:
      Date:
*/
#include <stdio.h>

// Function Declarations
void printOne (int x);

int main (void)
{
// Local Declarations
   int a;

// Statements
   a = 5;                                      // First call
   printOne (a);

   a = 33;
   printOne (a);                               // Second call
```

continued

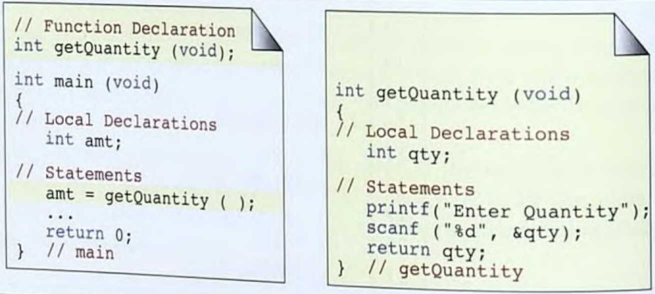
PROGRAM 4-2 *void* Function with a Parameter *(continued)*

```

   // Done. Return to operating system.
   return 0;
}  // main

/* ================== printOne ===================
   Print one integer value.
      Pre   x contains number to be printed
      Post  value in x printed
*/
void printOne (int x)
{
// Statements
   printf("%d\n", x);
   return;
}  // printOne
```

```
Results:
5
33
```

Non-void Functions without Parameters

Some functions return a value but don't have any parameters. The most common use for this design reads data from the keyboard or a file and returns the data to the calling program. We show this design in Figure 4-7. Note that the called function contains all of the code to read the data. In this simple example, only a prompt and a read are required. Later we introduce concepts that require much more code.



```
// Function Declaration
int getQuantity (void);

int main (void)
{
// Local Declarations
   int amt;

// Statements
   amt = getQuantity ( );
   ...
   return 0;
}  // main
```

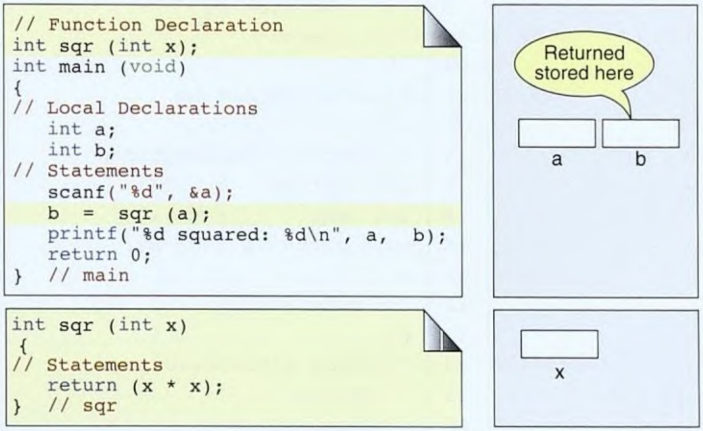
```
int getQuantity (void)
{
// Local Declarations
   int qty;

// Statements
   printf("Enter Quantity");
   scanf ("%d", &qty);
   return qty;
}  // getQuantity
```

FIGURE 4-7 Non-void Function without Parameters

Non-void Functions with Parameters

Figure 4-8 contains a function that passes parameters and returns a value—in this case, the square of the parameter. Note how the returned value is placed in the variable, `b`. This is not done by the call; it is a result of expression evaluation. Since the call is a postfix expression, it has a value—whatever is returned from the function. After the function has been executed and the value returned, the value on the right side of the assignment expression is the returned value, which is then assigned to `b`. Thus, again we see the power of expressions in the C language. Note that the function, `sqr`, has no side effect.

In *main*, the function call is evaluated first because it is a postfix expression, which has a higher precedence than the assignment expression. To evaluate it, the function call is executed. Its value is the value returned by the called function. The expression value is then used in the binary expression (assignment). The side effect of the binary expression stores the returned value in the variable.



```
// Function Declaration
int sqr (int x);
int main (void)
{
// Local Declarations
   int a;
   int b;
// Statements
   scanf("%d", &a);
   b  =  sqr (a);
   printf("%d squared: %d\n", a,  b);
   return 0;
}  // main
```

```
int sqr (int x)
 {
// Statements
   return (x * x);
}  // sqr
```

FIGURE 4-8 Calling a Function That Returns a Value

In Chapter 3 we saw that an expression becomes an expression statement when it is terminated by a semicolon. The function call, which is a postfix expression and is terminated by a semicolon, is an expression statement. Functions that do not return a value (*void* functions) can only be used in an expression statement as shown below.

```
printOne();                                   // See Figure 4-6
```

Functions that return a value can be used either in an expression statement or as an expression. When it is used as an expression statement, the return value is discarded. When it is used as an expression, as in Figure 4-8, its value is the value returned from the function. The following example demonstrates calling sqr as an expression statement and as an expression. Note that the expression statement is useless because neither the function call nor the statement has a side effect.

```
sqr (a);         // Expression Statement. No side effect
b = sqr(a);      // Return value is stored in b
```

In large programs, *main* is written with only function calls. To demonstrate how this would be done with our simple example, we combine our square and print functions into Program 4-3.

PROGRAM 4-3 Read a Number and Square It

```
/* This program reads a number and prints its square.
      Written by:
      Date:
*/
#include <stdio.h>

// Function Declarations
int  getNum   (void);
int  sqr      (int x);
void printOne (int x);

int main (void)
{
// Local Declarations
   int a;
   int b;

// Statements
   // Get number and square it
   a = getNum ();

   // Square the number just read
   b = sqr (a);

   // Now print it
   printOne(b);

   return 0;
} // main

/* ================== getNum ===================
```

continued

PROGRAM 4-3 Read a Number and Square It *(continued)*

```
   Read number from keyboard and return it.
      Pre   nothing
      Post  number read and returned
*/
int getNum (void)
{
// Local Declarations
   int numIn;

// Statements
   printf("Enter a number to be squared: ");
   scanf ("%d", &numIn);
   return numIn;
}  // getNum

/* ==================== sqr ====================
   Return the square of the parameter.
      Pre   x contains number to be squared
      Post  squared value returned
*/
int sqr (int x)
{
// Statements
   return (x * x);
}  // sqr

/* ==================== printOne ====================
   Print one integer value.
      Pre   x contains number to be printed
      Post  value in x printed
*/
void printOne (int x)
{
// Statements
   printf("The value is: %d\n", x);
   return;
}  // printOne
```

```
Results:
Enter a number to be squared: 81
The value is: 6561
```

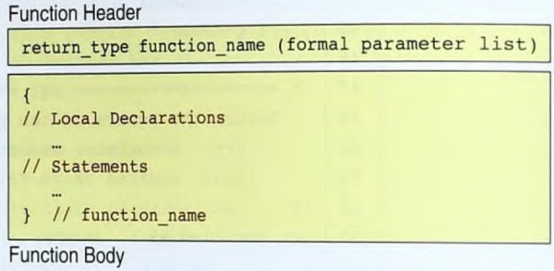
Program 4-3 Analysis Our simple program has grown to four functions, including *main.* This is an example of **decomposition**, the process of breaking a complex problem into simple parts. While our example is not really complex, it still demonstrates the concept.

We made one slight modification to `printOne` in this program. To make the output a little more meaningful, we added `The value is:` to the print statement. As a general principle, the person at the monitor should not have to guess what the output is.

Function Definition

Now that we have seen examples of the basic function formats, let's look at functions in more detail. We begin with the function definition.

The function definition contains the code for a function. It is made up of two parts: the function header and the function body, which is a compound statement. Remember that a compound statement must have opening and closing braces, and it has declaration and statement sections. The function definition format is shown in Figure 4-9.



```
return_type function_name (formal parameter list)
```

```
{
// Local Declarations
   …
// Statements
   …
}  // function_name
```

FIGURE 4-9 Function Definition

Function Header

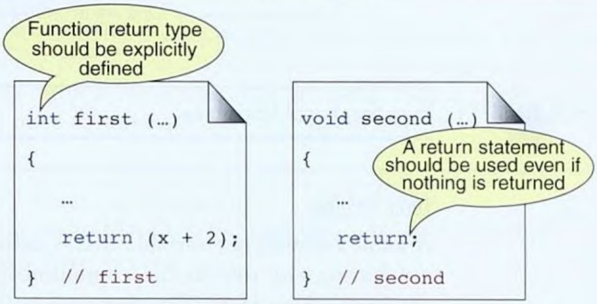
A **function header** consists of three parts: the return type, the function name, and the formal parameter list. A semicolon is not used at the end of the function definition header.

With the implementation of the C99 standard, the return type became a required element of the syntax. When nothing is to be returned, the return type is *void*.

Function Body

The **function body** contains the local declarations and the function statements. The body starts with local definitions that specify the variables needed by the function. After the local definitions, the function statements, terminating with a *return* statement, are coded. If a function *return* type is *void*, it can be written without a *return* statement. Because we believe that *default* statements should be explicitly coded for clarity, we strongly recommend that every function, even *void* functions, have a *return* statement.

Figure 4-10 shows two functions, `first` and `second`. The function `first` has been declared to return an integer value. Its *return* statement therefore contains the expression `x + 2`. When the *return* statement is executed, the expression is evaluated and the resulting value is returned. The function `second` returns nothing; its *return* type is *void*. It therefore needs no *return* statement—the end of the function acts as a *void return*. Again, we strongly recommend that you include a *return* statement even for *void* functions. In this case, the *return* statement has no expression; it is just completed with a semicolon.



```
int first (…)
{
   …
   return (x + 2);
}  // first
```

```
void second (…)
{
   …
   return;
}  // second
```

Function return type should be explicitly defined
A return statement should be used even if nothing is returned

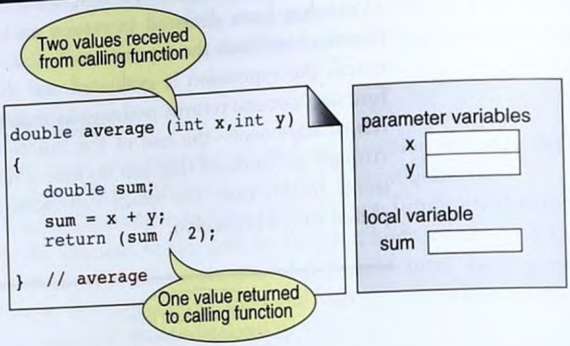
FIGURE 4-10 Function Return Statements

Formal Parameter List

In the definition of a function, the **parameters** are contained in the **formal parameter list**. This list defines and declares the variables that will contain the data received by the function. The parameter list is always required. If the function has no parameters—that is, if it does not receive any data from the calling function—then the fact that the parameter list is empty is declared with the keyword *void*.

In C each variable must be defined and declared fully with multiple parameters separated by commas. To make the parameter list easier to read when there are multiple lines, we align the parameter types and their names with tabs. Attention to these little details makes the code much easier to read and understand.

In Figure 4-11, the variables x and y are formal parameters that receive data from the calling function's parameters. Since they are value parameters, copies of the values being passed are stored in the called function's memory area. If the function changes either of these values, only the copies will be changed. The original values in the calling function remain unchanged.



```
double average (int x,int y)
{
   double sum;

   sum = x + y;
   return (sum / 2);

}  // average
```

Two values received from calling function
One value returned to calling function
parameter variables
x
y
local variable
sum

FIGURE 4-11 Function Local Variables

Local Variables

A **local variable** is a variable that is defined inside a function and used without having any role in the communication between functions. Figure 4-11 shows an example of a function with both formal parameters and a local variable, sum.

> Formal and Actual Parameters
>
> ***Formal parameters* are variables that are declared in the header of the function definition.**
>
> ***Actual parameters* are the expressions in the calling statement.**
>
> **Formal and actual parameters must match exactly in type, order, and number. Their names, however, do not need to match.**

Function Declaration

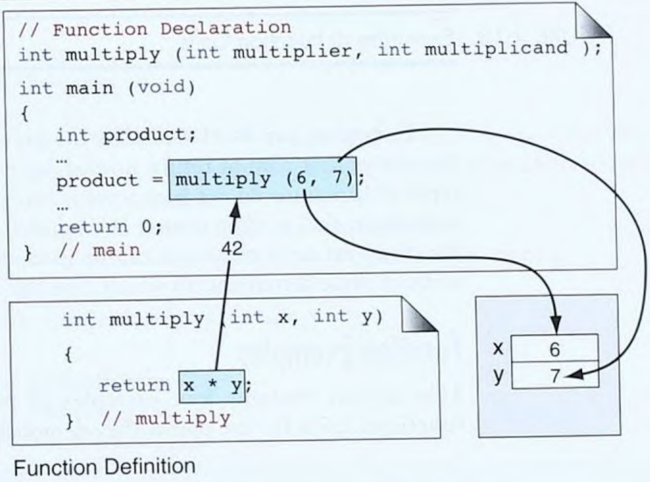
Function declarations consist only of a function header; they contain no code. Like function definition headers, function declaration headers consist of three parts: the return type, the function name, and the formal parameter list. Unlike the header for the function definition, function declarations are terminated with a semicolon.

The return type and the parameter list are required entries. If the program has no parameters, we code *void* in the parentheses. If the program has multiple parameters, we separate each type-identifier set with commas.

The C standard does not require identifier names for the function declaration's formal parameters. This does not prevent our using names in a function declaration, however. In fact, readability and understandability are improved if names are used. One point to note, however, is that the names do not need to be the same in the function declaration and the function definition; on the other hand, if the types are not the same, we get a compile error. The compiler checks the types in the function statements with the types in the call to ensure that they are the same or at least compatible. The major reason to include the identifiers in the function declaration is documentation; therefore, their names should be meaningful. Don't use generic identifiers such as a or x.

Declarations are placed in the global declaration section before *main*. Grouping all function declarations at the beginning of the program makes them available whenever they are needed. Function declarations also provide an excellent quick reference for functions used in the program, making them excellent documentation.

Figure 4-12 demonstrates several of these concepts. The function declaration tells the program that a function named `multiply`, which accepts two integers and returns one integer, will be called. That is all the compiler needs; it does not need to know anything else to evaluate the call.



```
// Function Declaration
int multiply (int multiplier, int multiplicand );
int main (void)
{
   int product;
   …
   product = multiply (6, 7);
   …
   return 0;
}  // main
```

```
   int multiply (int x, int y)
   {
      return x * y;
   }  // multiply
```

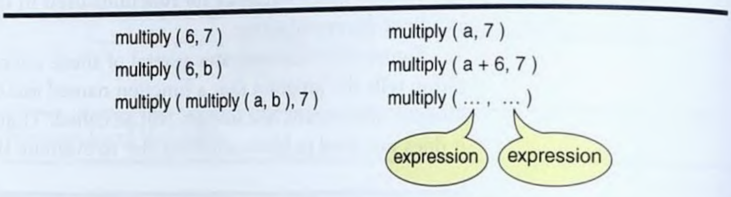
FIGURE 4-12 Parts of a Function Call

Figure 4-12 also demonstrates that the formal parameter names in the declaration do not need to be the same as the actual parameter names. In this case, the names in the function declarations are much more meaningful and therefore should have been used in the function definition.

The Function Call

A **function call** is a postfix expression with precedence 16. The operand in a function call is the function name; the operator is the parentheses set, (...), which contains the **actual parameters**. The actual parameters identify the values that are to be sent to the called function. They match the function's formal parameters in type and order in the parameter list. If the program has multiple actual parameters, they are separated by commas.

There are many different ways to call a function. In Figure 4-13, `multiply` is called six different ways. The first three show calls with primary expressions. The fourth uses a binary expression, `a + 6`, as the first parameter value, and the fifth shows the function `multiply (a, b)` as its own first parameter. The last example sums it all up: Any expression that reduces to a single value can be passed as a parameter.



```
multiply ( 6, 7 )                      multiply ( a, 7 )
multiply ( 6, b )                      multiply ( a + 6, 7 )
multiply ( multiply ( a, b ), 7 )      multiply ( ... , ... )
```

FIGURE 4-13 Examples of Function Calls

Functions can be classified by the presence or absence of a return value. Functions that cannot return a value have a return type of *void*. Since these types of functions do not have a value, they can be used only as a stand-alone statement; that is, they cannot be included as part of an expression. All other functions return a value and can be used either as part of an expression or as a stand-alone statement, in which case the value is simply discarded.

Function Examples

This section contains four examples of programs in which functions call functions. Look for the points they demonstrate.

EXAMPLE 4-1 Print Least Significant Digit

Program 4-4 prints the least significant (rightmost) digit of any integer read from the keyboard.

PROGRAM 4-4 Print Least Significant Digit

```
/* This program prints the first digits of an integer
   read from the keyboard
      Written by:
      Date:
*/
#include <stdio.h>

```

continued

PROGRAM 4-4 Print Least Significant Digit *(continued)*

```
// Function Declarations
int firstDigit (int num);

int main (void)
{
// Local Declarations
   int number;
   int digit;

// Statements
   printf("Enter an integer: ");
   scanf ("%d", &number);

   digit = firstDigit (number);
   printf("\nLeast significant digit is: %d\n", digit);

   return 0;
}  // main

/* ================ firstDigit =================
   This function extracts the least significant digit
   of an integer.
      Pre   num contains an integer
      Post  Returns least significant digit.
*/
int firstDigit (int num)
{
// Statements
   return (num % 10);
}  // firstDigit
```

```
Results:
Enter an integer: 27

Least significant digit is: 7
```

Program 4-4 Analysis This extremely simple program demonstrates how to call a function from *main*. Note that even though `firstDigit` is used only in *main*, its function declaration is at the global level. In the sample run, when `firstDigit` was executed, it returned 7, which was then put into `digit` and printed.

EXAMPLE 4-2 Add Two Digits

Let's write a function that extracts and adds the two least significant digits of any integer number. The design is shown in Figure 4-14. It consists of four functions, *main*, addTwoDigits, firstDigit, and secondDigit. The function addTwoDigits is called by *main*; addTwoDigits calls firstDigit and secondDigit. (Turn to Section 4.8, "Software Engineering," if you have trouble understanding the structure charts.)

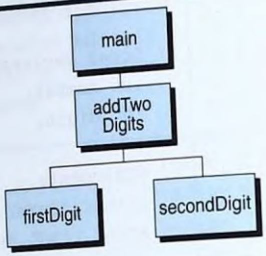


FIGURE 4-14 Design for Add Two Digits

The code is shown in Program 4-5.

PROGRAM 4-5 Add Two Digits

```
/* This program extracts and adds the two least
   significant digits of an integer.
      Written by:
      Date:
*/
#include <stdio.h>

//  Function Declarations
int addTwoDigits (int num);
int firstDigit   (int num);
int secondDigit  (int num);

int main (void)
{
// Local Declarations
   int  number;
   int  sum;

// Statements
   printf("Enter an integer: ");
   scanf ("%d", &number);

   sum = addTwoDigits (number);
   printf ("\nSum of last two digits is: %d", sum);
```

continued

PROGRAM 4-5 Add Two Digits *(continued)*

```

   return 0;
}  // main

/* ================== addTwoDigits ==================
    Adds the first two digits of an integer.
       Pre  num contains an integer
       Post returns sum of least significant digits
*/
int addTwoDigits (int number)
{
// Local Declarations
   int result;

// Statements
   result = firstDigit(number) + secondDigit(number);
   return result;
}  // addTwoDigits

/* ================== firstDigit =================
   Extract the least significant digit of an integer.
      Pre   num contains an integer
      Post  Returns least significant digit.
*/
int firstDigit (int num)
{
// Statements
   return (num % 10);
}  // firstDigit

/* ================= secondDigit =================
   Extract second least significant (10s) digit
      Pre   num is an integer
      Post  Returns digit in 10s position
*/
int secondDigit (int num)
{
// Local Declarations
   int  result;

// Statements
   result = (num / 10) % 10;
   return result;
}  // secondDigit
```

continued

PROGRAM 4-5 Add Two Digits *(continued)*

```
Results:
Run 1
   Enter an integer: 23
   Sum of last two digits is: 5

Run 2
   Enter an integer: 8
   Sum of last two digits is: 8
```

Program 4-5 Analysis A natural question asked by students when they first read this program is, "Why not put `firstDigit` and `secondDigit` as in-line code in `addTwoDigits`?" This may seem to be an obvious way to code the problem. And after all, each of the called functions is *only one statement.*

The answer is that, although each function is only one statement, it does a job that can be used in other places. One of the principles of structured programming is that processes should appear in a program in only one place. For example, we have used the same code for `firstDigit` that we used in the first program. If a function is to be reusable in this way, it must do only one thing. The short answer, then, is that it is better structured programming. It is the nature of the task to be performed, not the amount of code, that determines if a function should be used.

An interesting point to note is the way these two different digits were calculated. To get the least significant digit, we took the 10s modulus of the number. But to get the second digit, we had to divide by 10. Can you figure out how to sum the digits in a three-digit number? We will give you a chance to do this in the problems at the end of the chapter.

Note that we tested the program with two different numbers, one containing only one digit. It is often necessary to run the program with more than one test case. Another test case that should be run is a negative number. What do you think would happen? As a programmer, not only should you run several tests but you should predict the results before you run the program.

EXAMPLE 4-3 Format Long Integer

Program 4-6 reads a long integer and prints it with a comma after the first three digits, such as 123,456. The number is printed with leading zeros (see field specification `%03d` in Statement 40) in case the value is less than 100,000.

PROGRAM 4-6 Print Six Digits with Comma

```
1  /* This program reads long integers from the keyboard
2     and prints them with leading zeros in the form
3     123,456 with a comma between 3rd & 4th digit.
4        Written by:
5        Date:
```

continued

PROGRAM 4-6 Print Six Digits with Comma *(continued)*

```
*/
#include <stdio.h>
// Function Declarations
void printWithComma (long num);

int main (void)
{
// Local Declarations
   long  number;

// Statements
   printf("\nEnter a number with up to 6 digits: ");
   scanf ("%ld", &number);
   printWithComma (number);

   return 0;
}  // main

/* =============== printWithComma ================
   This function divides num into two three-digit
   numbers and prints them with a comma inserted.
      Pre   num is a six digit number
      Post  num has been printed with a comma inserted
*/
void printWithComma (long num)
{
// Local Declarations
   int  thousands;
   int  hundreds;

// Statements
   thousands  = num / 1000;
   hundreds   = num % 1000;

   printf("\nThe number you entered is \t%03d,%03d",
            thousands, hundreds);
   return;
}  // printWithComma
```

```
Results:
Run 1
   Enter a number with up to 6 digits: 123456

   The number you entered is       123,456
```

continued

PROGRAM 4-6 Print Six Digits with Comma *(continued)*

```
Run 2
   Enter a number with up to 6 digits: 12

   The number you entered is       000,012
```

Program 4-6 Analysis Once again, we have a simple program that has the makings of a very useful function. C has no built-in functions that will provide number formatting such as commas and dollar signs. We have to program these details. And since this logic will be used over and over again, it must be in its own function. Note, however, that more work is needed to print numbers less than 100,000 correctly. However, we have not yet introduced the tools to do the complete job.

Again, we have used two test cases to show that more work must be done. An even bigger problem occurs if we try to format a small negative number. Can you see what the problem is? If not, code the problem and run it to see. We will explain how to handle these problems in Chapter 5.

EXAMPLE 4-4 Print Tuition for Strange College

Our next example calculates and prints the annual tuition for a student enrolled in Strange College. In this college, students can take an unlimited number of units each term. Each term, the students are charged $10 per unit plus a $10 registration fee. To discourage them from overloading, the college charges $50 extra for each 12 units, or fraction thereof, a student takes after the first 12 units. For example, if a student takes 13 units, the tuition is $190 ($10 for registration, plus 13 times $10 for units, plus a $50 penalty for the one extra unit). If a student takes 25 units, the tuition is $360 ($10 for registration, plus 25 times $10 for units, plus $100 for two penalty fees). The design for this problem is shown in Figure 4-15.

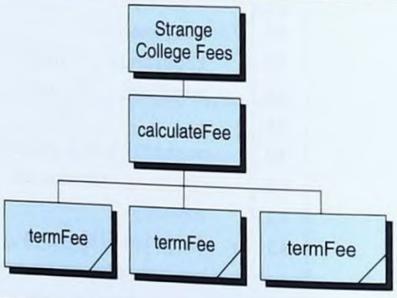


FIGURE 4-15 Design for Strange College fees

The code for Strange College is shown in Program 4-7.

PROGRAM 4-7 Strange College Fees

```
/* This program prints the tuition at Strange College.
   Strange charges $10 for registration, plus $10 per
   unit and a penalty of $50 for each 12 units, or
   fraction of 12, over 12.
      Written by:
      Date:
*/
#include <stdio.h>

#define REG_FEE    10
#define UNIT_FEE   10
#define EXCESS_FEE 50

// Function Declarations
int calculateFee (int firstTerm, int secondTerm,
                  int thirdTerm);
int termFee      (int units);

int main (void)
{
// Local Declarations
   int  firstTerm;
   int  secondTerm;
   int  thirdTerm;
   int  totalFee;

// Statements
   printf("Enter units for first term:  ");
   scanf ("%d", &firstTerm);

   printf("Enter units for second term: ");
   scanf ("%d", &secondTerm);

   printf("Enter units for third term:  ");
   scanf ("%d", &thirdTerm);

   totalFee = calculateFee
              (firstTerm, secondTerm, thirdTerm);
   printf("\nThe total tuition is : %8d\n", totalFee);

   return 0;
}  // main

```

continued

PROGRAM 4-7 Strange College Fees *(continued)*

```
/* ================ calculateFee =================
   Calculate the total fees for the year.
      Pre   The number of units to be taken each term.
      Post  Returns the annual fees.
*/
int calculateFee (int firstTerm, int secondTerm,
                  int thirdTerm)
{
// Local Declarations
   int  fee;

// Statements
   fee = termFee   (firstTerm)
       + termFee (secondTerm)
       + termFee (thirdTerm);
   return fee;
}  // calculateFee

/* =================== termFee ====================
   Calculate the tuition for one term
      Pre   units contains units for the term
      Post  The fee is calculated and returned
*/
int termFee (int units)
{
// Local Declarations
   int  totalFees;

// Statements
   totalFees = REG_FEE
             + ((units - 1)/12 * EXCESS_FEE)
             + (units * UNIT_FEE);
   return (totalFees);
}  // termFee
```

```
Results:
Enter units for first term:  10
Enter units for second term: 20
Enter units for third term:  30

The total tuition is :     780
```

Program 4-7 Analysis The most interesting aspect of Program 4-7 is how we call `termFee` three different times in one function. Let's look at how it works. The key statement follows.

```
fee = termFee   (firstTerm)
    + termFee (secondTerm)
    + termFee (thirdTerm);
```

A function call is a postfix expression. Therefore, it evaluates from the left. To evaluate the expression (three function calls) on the right of the assignment operator, we first evaluate the first expression, the call to `termFee` with the number of units for the first term. When `termFee` completes the first time, the return value (110) replaces the call. At this point, we have the expression shown below.

```
110 + termFee (secondTerm) + termFee (thirdTerm)
```

When `termFee` is executed a second time, its return value (260) becomes the value of the second expression and we have

```
110 + 260 + termFee (thirdTerm)
```

After the third call to `termFee`, the expression on the right of the assignment operator is ready for evaluation. Its value is 780, which is assigned to `fee`.

At least two more tests are needed to completely evaluate this program. We would run it with all three terms having 0 units and then do another test of 11, 12, and 13 units.

4.4 Inter-Function Communication

Although the calling and called function are two separate entities, they need to communicate to exchange data. The data flow between the calling and called functions can be divided into three strategies: a *downward flow* from the calling to the called function, an *upward flow* from the called to the calling function, and a *bi-directional flow* in both directions. Figure 4-16 shows the flow of data in these cases.

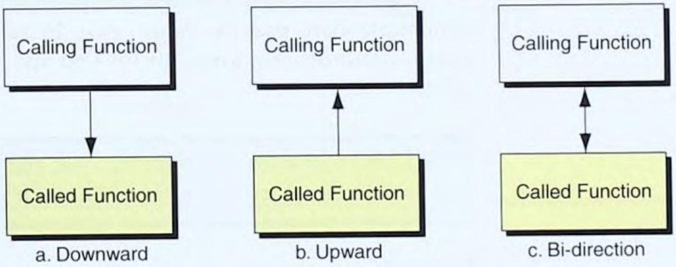


FIGURE 4-16 Data Flow Strategies

Basic Concept

Let us first discuss the general concept before we discuss how they are done in C.

Downward Flow

In **downward communication**, the calling function sends data to the called function. No data flows in the opposite direction. In this strategy, copies of the data items are passed from the calling function to the called function. The called function may change the values passed, but the original values in the calling function remain untouched. An example of this type of communication is passing data to a print function.

Upward Flow

Upward communication occurs when the called function sends data back to the called function without receiving any data from it. A good example of this type of communication is when the called function reads data from the keyboard that needs to be passed back to the called function.

Bi-directional Flow

Bi-directional communication occurs when the calling function sends data down to the called function. During or at the end of its processing, the called function then sends data up to the calling function. For example, the calling function may send data to the called function, which it manipulates and sends up to the calling function.

C Implementation

Now let us examine how these three types of communication are implemented in C. In the discussion that follows, it is important to remember that the flow refers to the data. The data flows down to the called function. The data flows from the called function back to the calling function.

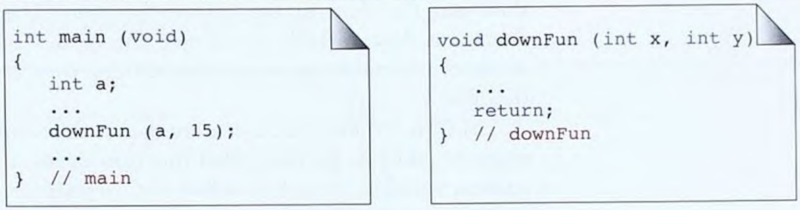
Most programming languages have three strategies for inter-function communication: **pass by value**, **pass by reference**, and return. The C language, unfortunately, uses only the first and last strategies; there is no pass by reference in C.

> The C language uses only pass by value and return to achieve three types of communications between a calling and a called function.

Downward Communication

The pass-by-value mechanism in C is a perfect solution for the communication in the downward direction. A variable is declared and defined in the called function for each value to be received from the calling function. The calling function sends a copy of each value to the called function; no data flows upward.

Figure 4-17 demonstrates downward communication. Two data items are passed from *main* to the *downFun* function. One data value is a literal, the other is the value of a variable.

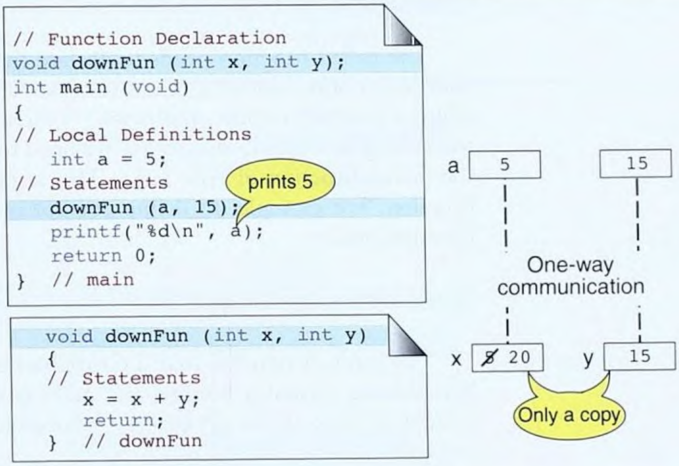


```
int main (void)
{
   int a;
   ...
   downFun (a, 15);
   ...
}  // main
```

```
void downFun (int x, int y)
{
   ...
   return;
}  // downFun
```

FIGURE 4-17 Downward Communication in C

Downward communication is one-way communication. The calling function can send data to the called function, but the called function cannot send any data to the calling function. Figure 4-18 demonstrates one-way communication.



```
// Function Declaration
void downFun (int x, int y);
int main (void)
{
// Local Definitions
   int a = 5;
// Statements
   downFun (a, 15);
   printf("%d\n", a);
   return 0;
}  // main
```

```
   void downFun (int x, int y)
   {
   // Statements
      x = x + y;
      return;
   }  // downFun
```

FIGURE 4-18 Downward Communication

Upward Communication

C provides only one upward direction flow, the return statement. While it works well, only one data item can be returned. The only way that a called function can pass multiple data items up to the calling function is to access variables in the calling function and deposit the data there. However, C does not allow us to directly reference a variable in the calling function; in other words, we cannot access a variable in the calling function by its identifier. Therefore, we need another strategy to solve the problem.

The solution is for the calling function to pass the address of the variable to the called function. Given the variable's address, the called function can then put the data in the calling function. The calling function needs to declare a data variable to receive the data. The called function needs to declare a variable to store the address that it receives from the calling function.

In C, a variable that can store data is different from the variable that can store an address. So the called function needs a special type of variable, an address variable, or as it is called in C, a **pointer variable**, that points to the variable in the called function.

The calling function needs to pass the address of a variable to receive the data from the called function. This action is a pass-by-value mechanism although what is passed is not data, it is an address.

To get the address of a variable, we use the address operator (`&`). If the name of the variable in the calling function is `x`, we extract its address using the address operator (`&x`) and pass it as a parameter in the function call. The following example passes the addresses of `a` and `b` to a function named `upFun`.

```
upFun (&a, &b);
```

The called function needs to declare that the parameter is to receive an address; in other words, it needs to create an address variable. To declare an address variable, we use an *asterisk* (`*`) after the type. In other words, if `x` in the calling function is of type *int*, we need to declare and define a variable in the called function of type `int*`. This is done in the header of the called function. For example, to define our `upFun` function, we use the following function header.

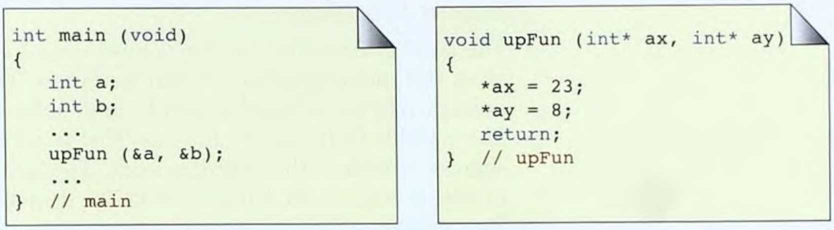
```
void upFun (int* ax, int* ay)
```

The asterisk signifies that the variables `ax` and `ay` are not data variables but address variables holding the address of *int* variables. Note that the asterisk as used in the declaration belongs to the type not the variables (`ax` and `ay`).

To change the data in the calling function, we need to use the variable's address parameter. Changing data through an address variable is known as indirect access; that is, we access the variable indirectly through its address stored in the parameter list and then change the data. For this reason, the asterisk is known as the **indirection operator**. To change the data, we again use an asterisk. This time, the asterisk is put immediately in front of the address variable's name. For example, to change the variable in the calling function to 23, we would use the following statement.

```
*ax = 23;
```

It is important to remember that these two uses of the asterisk, declaring an address variable and indirectly accessing the data, play completely different roles. The first asterisk, in the declaration, belongs to the type; the second asterisk is an expression that indirectly accesses the variable in the called program as shown in Figure 4-19.

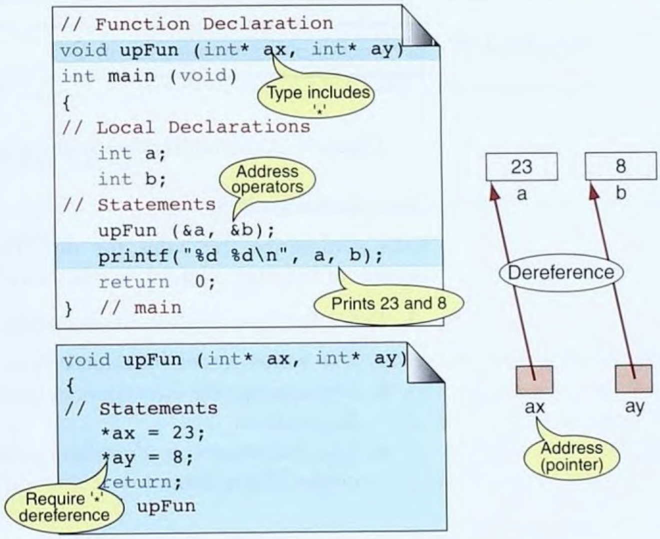


```
int main (void)
{
   int a;
   int b;
   ...
   upFun (&a, &b);
   ...
}  // main
```

```
void upFun (int* ax, int* ay)
{
   *ax = 23;
   *ay = 8;
   return;
}  // upFun
```

FIGURE 4-19 Upward Communication in C

We need to emphasize that when only one data item needs to be returned, we use the standard return statement. Only use upward communication when multiple items need to be returned. Figure 4-20 demonstrates the techniques for upward communication.



```
// Function Declaration
void upFun (int* ax, int* ay)
int main (void)
{
// Local Declarations
   int a;
   int b;
// Statements
   upFun (&a, &b);
   printf("%d %d\n", a, b);
   return 0;
}  // main
```

```
void upFun (int* ax, int* ay)
{
// Statements
   *ax = 23;
   *ay = 8;
   return;
}  // upFun
```

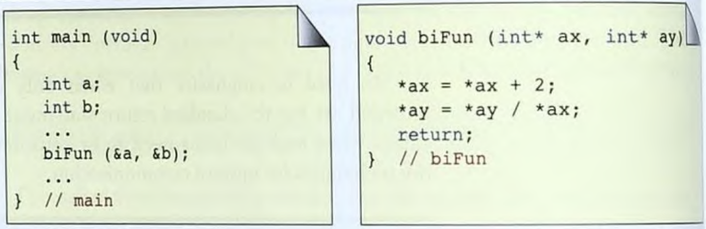
FIGURE 4-20 Upward Communication

To send data from the called function to the calling function:

1. We need to use the & symbol in front of the data variable when we call the function.
2. We need to use the * symbol after the data type when we declare the address variable
3. We need to use the * in front of the variable when we store data indirectly.

Bi-direction Communication

The strategy described for the upward direction can easily be augmented to allow the communication in both directions. The only difference is that the indirect reference must be used in both sides of the assignment statement. The variable in the called function first is accessed for retrieving using the address variable in the right-hand side. The same parameter is accessed again to store a value in the left-hand side. We demonstrate bi-directional access in Figure 4-21.



```
int main (void)
{
   int a;
   int b;
   ...
   biFun (&a, &b);
   ...
}  // main
```

```
void biFun (int* ax, int* ay)
{
   *ax = *ax + 2;
   *ay = *ay / *ax;
   return;
}  // biFun
```

FIGURE 4-21 Bi-directional Communication in C

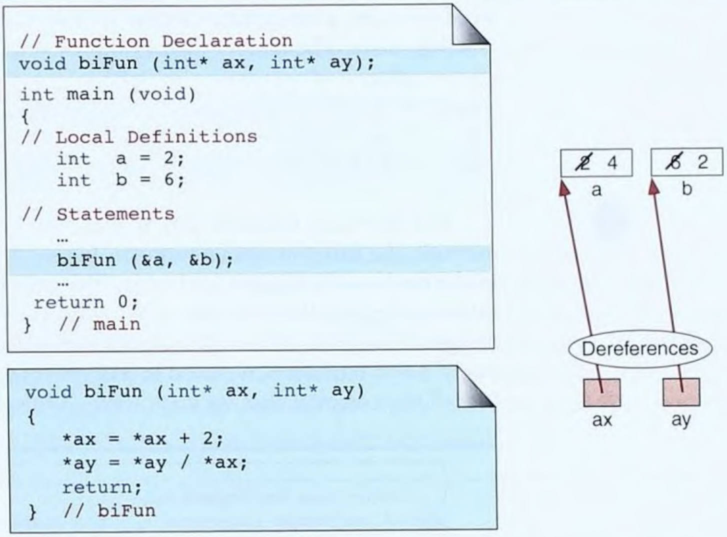
Figure 4-22 demonstrates bi-directional communication flow.

Communication Summary

Let's summarize the rules for inter-function communication. Like many aspects of learning a language, they must be memorized.

1. Rules for downward communication
   a. Use values in the function call to pass data.
   b. Use appropriate data types in the function parameter list to receive the data values.
   c. Use the parameter identifiers in the called function to access the local copies of the data.

2. Rules for upward and bi-directional communication
    a. Use `&variableName` in the function call to pass a reference to the variable.
    b. Use `type*` in the function parameter list to receive the variable's address.
    c. Use `*parameterName` in the function to reference the original variable.



```
// Function Declaration
void biFun (int* ax, int* ay);
int main (void)
{
// Local Definitions
   int  a = 2;
   int  b = 6;
// Statements
   ...
   biFun (&a, &b);
   ...
 return 0;
}  // main
```

```
void biFun (int* ax, int* ay)
{
   *ax = *ax + 2;
   *ay = *ay / *ax;
   return;
}  // biFun
```

FIGURE 4-22 Bi-directional Communication

Communication Examples

In this section we write some short programs to demonstrate function communication.

EXAMPLE 4-5 Exchange Function

Let's look at a program that uses the indirection operator to dereference data. One common process that occurs often in programming is exchanging two pieces of data. Let's write a function that, given two integer variables, exchanges them. Since two variables are being changed, we cannot use the *return* statement. Instead we pass addresses to make the changes.

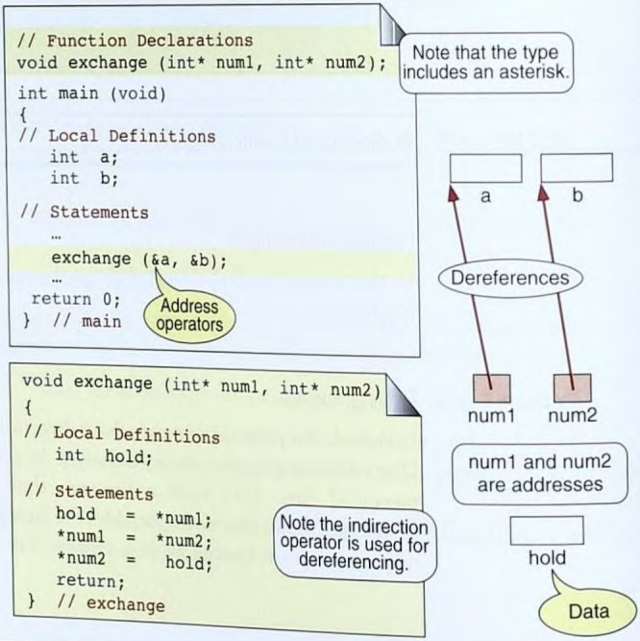
First, make sure you understand how to exchange two variables: We cannot simply assign variables to each other, as shown below.

```
                                          // This won't work
x = y;                                    // Result is y in both
y = x;
```

If you carefully trace these two statements, you will see that the original value of y ends up in both variables. Therefore, to exchange variables, we need to create a temporary variable to hold the first value while the exchange is being made. The correct logic is shown below.

```
hold = y;                                 // value of y saved
y    = x;                                 // x now in y
x    = hold;                              // original y now in x
```

The exchange function and its data flow are shown in Figure 4-23. First, examine the function declarations in Figure 4-23 carefully. Note the asterisk in the declaration of num1 and num2. The asterisk is used with the type declaration to specify that the type is an address—in this case, the address of an integer. Now, look at the call statement in *main*. Since we will change the values of a and b in *main*, we need to pass their addresses. The address operators (&) tell the compiler that we want their addresses passed, not their values.



```
// Function Declarations
void exchange (int* num1, int* num2);
int main (void)
{
// Local Definitions
   int  a;
   int  b;

// Statements
   ...
   exchange (&a, &b);
   ...
 return 0;
}  // main
```

```
void exchange (int* num1, int* num2)
{
// Local Definitions
   int  hold;

// Statements
   hold  =  *num1;
   *num1  =  *num2;
   *num2  =   hold;
   return;
}  // exchange
```

Note that the type includes an asterisk.
a
b
Dereferences
Address operators
num1
num2
num1 and num2 are addresses
Note the indirection operator is used for dereferencing.
hold
Data

FIGURE 4-23 Exchange Function

Now, look at the statements in exchange. First, we copy num1's value to hold. Since hold is a local variable, it is treated the same as any local variable—no special operators are required. However, num1 contains the address of the data we want, not the data itself. To get the value it is referring to, we must dereference it. This is done with the indirection operator, the asterisk (*). The statement is shown below.

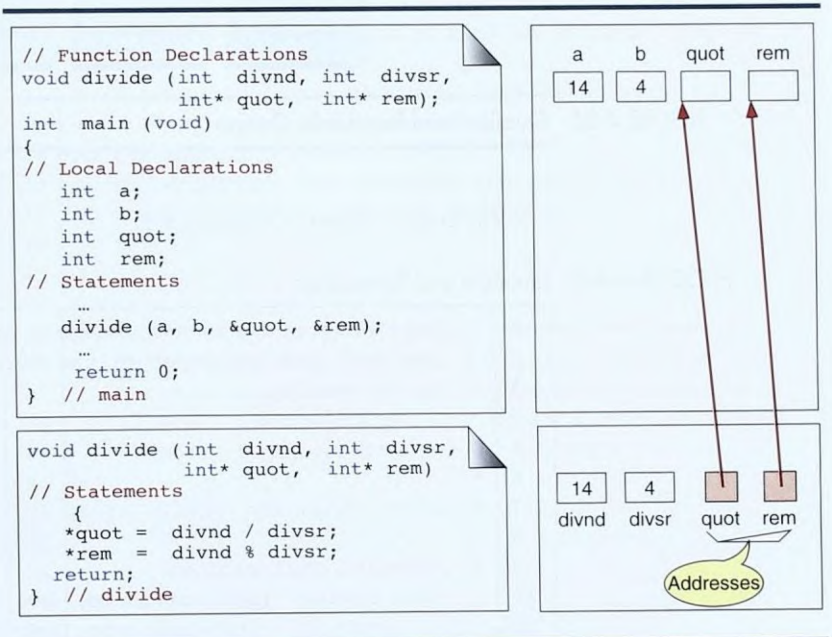
```
hold = *num1;
```

In the same manner, we can now dereference num2 and copy it to num1 and then complete the exchange by copying hold to num2. These statements are seen below.

```
*num1 = *num2;
*num2 =  hold;
```

Note that with the exception of hold, all of the data movement is done in the calling program's area. This is the power of the indirection operator.

Here is another simple example, one that uses both pass-by-value and pass-by-address parameters. We need to write a function that, given two numbers, calculates both the quotient and the remainder. Since we can't return two values, we will pass the addresses where we want the quotient and remainder stored. This problem is shown in Figure 4-24.



```
// Function Declarations
void divide (int  divnd, int  divsr,
             int* quot,  int* rem);
int  main (void)
{
// Local Declarations
   int  a;
   int  b;
   int  quot;
   int  rem;
// Statements
    …
   divide (a, b, &quot, &rem);
    …
    return 0;
}  // main
```

```
void divide (int  divnd, int  divsr,
             int* quot,  int* rem)
// Statements
   {
   *quot =  divnd / divsr;
   *rem  =  divnd % divsr;
  return;
}  // divide
```

FIGURE 4-24 Calculate Quotient and Remainder

Let's examine `divide` first. Note that the first two parameters are pass by value. You can tell this because their types are just *int*; there are no asterisks indicating that they are addresses. The last two parameters are addresses. Their types are `int *`; therefore, any references to them in the function must use the indirection operator.

Now look at the call in *main*. Note that `a` and `b` are simply passed by using their identifiers. They are primary expressions whose value is the contents of the variable. On the other hand, `quot` and `rem` are passed as addresses by using the address operator. In this case, the value of the primary expression created by the address operator and the identifier is the address of the variable. Therefore, we pass an address value rather than a data value.

As the programmer, it is our job to know what needs to be passed, a value or an address, when we write a call. Similarly, when we use the parameters in the called program, we must remember to use the indirection operator when we have an address.

Let's use `divide` in a program that reads two integers, divides them, and then prints the quotient and the remainder. The design for this program is shown in Figure 4-25.

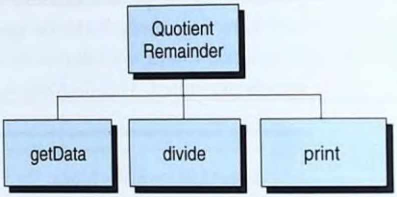


FIGURE 4-25 Quotient and Remainder Design

The code is shown in Program 4-8.

PROGRAM 4-8 Quotient and Remainder

```
/* This program reads two integers and then prints the
   quotient and remainder of the first number divided
   by the second.
      Written by:
      Date:
*/
#include <stdio.h>

// Function Declarations
   void divide (int  dividend, int  divisor,
                     int* quotient, int* remainder);
```

continued

PROGRAM 4-8 Quotient and Remainder *(continued)*

```
   void getData (int* dividend, int* divisor);
   void print   (int  quotient, int  remainder);

int main (void)
{
// Local Declarations
int  dividend;
int  divisor;
int  quot;
int  rem;

// Statements
   getData (&dividend, &divisor);
   divide  (dividend,  divisor, &quot, &rem);
   print   (quot, rem);

   return 0;
}  // main

/* ================== getData ==================
   This function reads two numbers into variables
   specified in the parameter list.
      Pre   Nothing.
      Post  Data read and placed in calling function.
*/
void getData  (int* dividend, int* divisor)
{
// Statements
   printf("Enter two integers and return: ");
   scanf ("%d%d", dividend, divisor);
   return;
}  // getData

/* =================== divide ====================
   This function divides two integers and places the
   quotient/remainder in calling program variables
      Pre   dividend & divisor contain integer values
      Post  quotient & remainder calc'd
*/
void divide (int  dividend, int  divisor,
             int* quotient, int* remainder)
{
// Statements
   *quotient  = dividend / divisor;
```

continued

PROGRAM 4-8 Quotient and Remainder *(continued)*

```
56       *remainder = dividend % divisor;
57       return;
58    }  // divide
59
60    /* =================== print ===================
61       This function prints the quotient and the remainder
62          Pre   quot contains the quotient
63                rem contains the remainder
64          Post  Quotient and remainder printed
65    */
66    void print (int quot, int rem)
67    {
68    // Statements
69       printf ("Quotient : %3d\n", quot);
70       printf ("Remainder: %3d\n", rem);
71       return;
72    }  // print
```

Program 4-8 Analysis First, look at the design of this program. Note how *main* contains only calls to subfunctions. It does no work itself; like a good manager, it delegates all work to lower levels in the program. This is how you should design your programs.

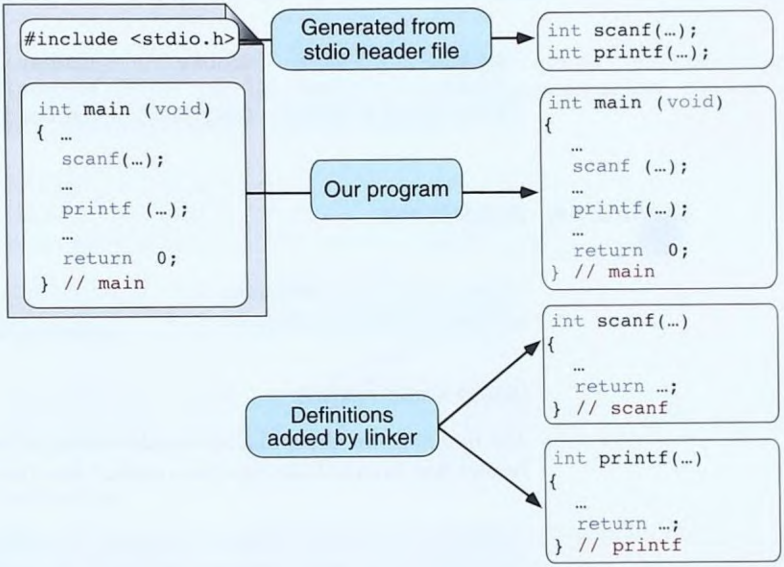
Study the `getData` function carefully. First note that the parameters identify the variables as addresses. Verify that they contain an asterisk as a part of their type. Now look at the *scanf* statement carefully. What is missing? There are no address operators for the variables! We don't need the address operators because the parameters are already addresses. We have used the address operator (`&`) up to this example to tell the compiler that we want the address of the variable, not its contents. Since the parameters were passed as addresses (see Statement 24), we don't need the address operator.

Now study the way we use `quotient` and `remainder` in `divide`. We pass them as addresses by using the address operator. Since we are passing addresses, we can change their values in *main*. In the function definition, the formal parameters show these two parameters to be addresses to integers by showing the type as `int *` (see Statement 52). To change the value back in *main*, we use the indirection operator when we assign the results of the divide and modulo operations.

4.5 Standard Functions

C provides a rich collection of standard functions whose definitions have been written and are ready to be used in our programs. To use these functions, we must include their function declarations. The function declarations for these functions are grouped together and collected in several header files. Instead of adding the individual declarations of each function, therefore, we simply include the headers at the top of our programs. Appendix F contains the function declarations for most standard functions.

Figure 4-26 shows how two of the C standard functions that we have used several times are brought into our program. The *include* statement causes the library header file for standard input and output (*stdio.h*) to be copied into our program. It contains the declarations for *printf* and *scanf*. Then, when the program is linked, the object code for these functions is combined with our code to build the complete program.



```
#include <stdio.h>

int main (void)
{ …
   scanf(…);
   …
   printf (…);
   …
   return  0;
} // main
```

```
int scanf(…);
int printf(…);
```

```
int main (void)
{
   …
   scanf (…);
   …
   printf(…);
   …
   return  0;
} // main
```

```
int scanf(…)
{
   …
   return …;
} // scanf
```

```
int printf(…)
{
   …
   return …;
} // printf
```

FIGURE 4-26 Library Functions and the Linker

Math Functions

Many important library functions are available for mathematical calculations. Most of the function declarations for these functions are in either the math header file (*math.h*) or standard library (*stdlib.h*). In general, the integer functions are found in *stdlib.h*.

Absolute Value Functions

The functions described in this section return the absolute value of a number. An **absolute value** is the positive rendering of the value regardless of its sign. There are three integer functions and three real functions.

The integer functions are *abs*, *labs*, and *llabs*. For *abs* the parameter must be an *int* and it returns an *int*. For *labs* the parameter must be a *long int*, and it returns a *long int*. For *llabs* the parameter must be a *long long int*, and it returns a *long long int*.

```
int       abs   (int       number);        // stdlib.h
long      labs  (long      number);        // stdlib.h
long long llabs (long long number);        // stdlib.h
```

The real functions are *fabs*, *fabsf*, and *fabsl*. For *fabs* the parameter is a *double*, and it returns a *double*. The *float* version is named *fabsf* and the *long double* version is named *fabsl*.

```
double      fabs  (double      number);    // math.h
float       fabsf (float       number);    // math.h
long double fabsl (long double number);    // math.h
```

EXAMPLE 4-6 Absolute Numbers

```
abs  (3)      returns 3
fabs (-3.4)   returns 3.4
```

Complex Number Functions

The functions for manipulating complex numbers are collected in the *complex.h* header file. Some of the complex number functions are shown below.

```
double      cabs   (double complex      number);   // absolute
float       cabsf  (float complex       number);
long double cabsl  (long double complex number);
double      carg   (double complex      number);
float       cargf  (float complex       number);  // argument
long double cabsl  (long double complex number);
double      creal  (double complex      number);   // real
float       crealf (float complex       number);
long double creall (long double complex number);
double      cimag  (double complex      number);   // imaginary
float       cimagf (float complex       number);
long double cimagl (long double complex number);
```

Ceiling Functions

A **ceiling** is the smallest integral value greater than or equal to a number. For example, the ceiling of 3.0000001 is 4. If we consider all numbers as a continuous range from minus infinity to plus infinity (Figure 4-27), this function moves the number right to an integral value.

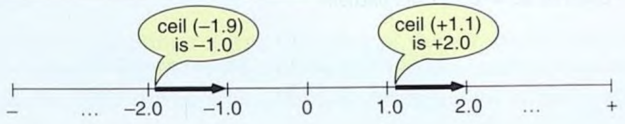


FIGURE 4-27 Ceiling Function

Although the ceiling functions determine an integral value, the return type is defined as a real value that corresponds to the argument. The ceiling function declarations are

```
double      ceil  (double      number);
float       ceilf (float       number);
long double ceill (long double number);
```

EXAMPLE 4-7 Ceiling Function

```
ceil (-1.9)   returns -1.0
ceil ( 1.1)   returns  2.0
```

Floor Functions

A **floor** is the largest integral value that is equal to or less than a number (see Figure 4-28). For example, the floor of 3.99999 is 3.0. Again, looking at numbers as a continuum, this function moves the number left to an integral value.

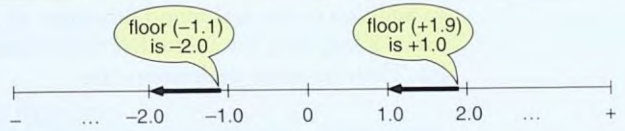


FIGURE 4-28 Floor Function

The floor function declarations are

```
double      floor  (double      number);
float       floorf (float       number);
long double floorl (long double number);
```

EXAMPLE 4-8 Floor Function

```
floor (-1.1)  returns -2.0
floor ( 1.9)  returns  1.0
```

Truncate Functions

The truncate functions return the integral in the direction of 0. They are the same as floor function for positive numbers and the same as ceiling function for negative numbers. Their function declarations are

```
double      trunc  (double      number);
float       truncf (float       number);
long double truncl (long double number);
```

EXAMPLE 4-9 Truncate Function

```
trunc (-1.1)  returns -1.0
trunc ( 1.9)  returns  1.0
```

Round Functions

The round functions return the nearest integral value. Their function declarations are

```
double      round  (double      number);
float       roundf (float       number);
long double roundl (long double number);
```

In addition to the real round functions, C provides two sets that return a *long int* or a *long long int*. Note that there is not a round function that returns an *int*. Their function declarations are

```
long int      lround   (double      number);
long int      lroundf  (float       number);
long int      lroundl  (long double number);
long long int llround  (double      number);
long long int llroundf (float       number);
long long int llroundl (long double number);
```

EXAMPLE 4-10 Round Function

```
round (-1.1)  returns -1.0
round ( 1.9)  returns  2.0
round (-1.5)  returns -2.0
```

Power Function

The power (*pow*) function returns the value of the x raised to the power y—that is, x^y. An error occurs if the base (x) is negative and the exponent (y) is not an integer, or if the base is zero and the exponent is not positive. The power function declarations are

```
double      pow   (double      n1, double      n2);
float       powf  (float       n1, float       n2);
long double powlf (long double n1, long double n2);
```

EXAMPLE 4-11 Power Function

```
pow (3.0, 4.0)  returns 81.0
pow (3.4, 2.3)  returns 16.687893
```

Square Root Function

The square root functions return the non-negative square root of a number. An error occurs if the number is negative. The square root function declarations are

```
double      sqrt   (double      n1);
float       sqrtf  (float       n1);
long double sqrtlf (long double n1);
```

EXAMPLE 4-12 Square Root Function

```
sqrt (25)  returns 5.0
```

Random Numbers

A random number is a number selected from a set in which all members have the same probability of being selected. Random numbers are useful in many areas of computer science. Two examples are application testing and gaming.

Random Number Generation

Because of their usefulness, most languages provide functions to generate random numbers. Each time the function is called, it generates another number. The function that generates random numbers uses a formula that has been mathematically designed to ensure that each number in the set in fact has the same probability of being selected; in other words, to ensure that each number is truly random. As a part of the formula, the function uses the previous random number. The design is shown in Figure 4-29.

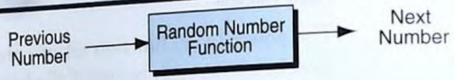


FIGURE 4-29 Random Number Generation

Now that we understand the basic design of a random number generator, a logical question is "How is the first random number generated?" The answer is from a seed. In C, we can use the default seed provided by the system (1) or we can specify our own seed.

Because C uses only one algorithm to generate random numbers, the series it produces is a **pseudorandom number series**. Only by using different seeds can we generate different random series. Calling the random function several times generates a series of random numbers as shown in Figure 4-30.

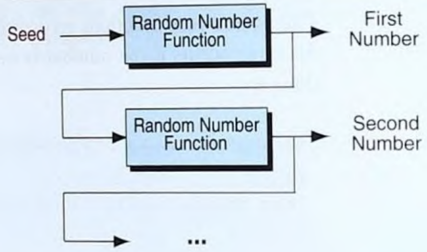


FIGURE 4-30 Generating a Random Number Series

We can create the same series or a different series of random numbers each time we run our program. Some programs require that the series be the same each time the program runs; others require that the series be different each time. We control the type of series by our choice of seeds.

Random Numbers in C

C provides two functions to build a random number series, seed random (*srand*) and random (*rand*). These functions are found in *stdlib.h*.

Seed Random Number Function

The seed random function, (*srand*), creates the starting seed for a number series. Its function declaration is

```
void srand (unsigned int seed);
```

EXAMPLE 4-13 Generate the same series in each run

To generate the same number series in each run, we can either omit *srand* or we can provide a constant seed random, preferably a prime number, such as 997.

```
srand (997);
```

EXAMPLE 4-14 Generate a different series in each run

To generate different series in each run, we use the time of day as the seed. The C call for the time, which requires the *time.h* library, is used in the following example.

```
srand(time (NULL));
```

Whichever series we need, *srand* should be *called only once* for each random number series, usually only once in a program.

***srand* must be called only once for each random number series.**

Random Number Function

The random number function, (*rand*) returns a pseudorandom integer between 0 and `RAND_MAX`, which is defined in the standard library as the largest number that *rand* can generate. The C standard requires that it be at least 32,767. Each call generates the next number in a random number series. The random number function declaration is

```
int rand (void);
```

EXAMPLE 4-15 Program to create a different series in each run

Program 4-9 is a simple program that prints three random numbers. Because we seed the random number generator with the time of day, each program execution generates a different series. We have run it twice to demonstrate that the random numbers are different.

PROGRAM 4-9 Creating Temporal Random Numbers

```
/* Demonstrate the use of the time function to generate
   a temporal random number series.
      Written by:
      Date:
*/
#include <stdio.h>
```

continued

PROGRAM 4-9 Creating Temporal Random Numbers *(continued)*

```
#include <stdlib.h>
#include <time.h>

int main (void)
{
// Statements
   srand(time(NULL));                          // Seed temporally

   printf("%d\n", rand());
   printf("%d\n", rand());
   printf("%d\n", rand());

   return 0;
}  // main
```

```
Results:
First Run
   9641
   16041
   6350

Second Run
   31390
   31457
   21438
```

EXAMPLE 4-16 Program to create the same series

Program 4-10 is a simple program that prints three random numbers. Because we seed the random number generator with a constant, each program execution generates the same series. We would have also generated the same series if we didn't use *srand*. We have run it twice to demonstrate that the random numbers are the same.

PROGRAM 4-10 Creating Pseudorandom Numbers

```
/* Demonstrate the use of the srand function to
   generate a pseudorandom number series.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

```

continued

PROGRAM 4-10 Creating Pseudorandom Numbers *(continued)*

```
int main (void)
{
// Statements
   srand(997);

   printf("%d\n", rand());
   printf("%d\n", rand());
   printf("%d\n", rand());

   return 0;
}  // main
```

```
Results:
First Run
   10575
   22303
   4276
Second Run
   10575
   22303
   4276
```

Using Random Numbers

The numbers returned by the random number generator are in a large range. The C standard specifies that it must be at least 0 to 32,767. When we need a random number in a different range, we must map the standard range into the required range. Two common ranges are a real-number series, usually 0.0 to 1.0,[2] and an integral range, such as 0 to 25 or 11 to 20.

Generating Random Integrals

To generate a random integral in a range *x* to *y*, we must first scale the number and then, if *x* is greater than 0, shift the number within the range. We scale the number using the modulus operator. For example, to produce a random number in the range 0…50, we simply scale the random number as shown below. Note that the scaling factor must be one greater than the highest random number needed.

```
rand ( ) % 51
```

Modulus works well when our range starts at 0. But what if we need a different range? In that case, we must shift the result. For example, suppose

[2] Note that 0.0 is included in the range but 1.0 is not. The maximum number is 0.99999999.

we want a random number between 3 and 7 (Figure 4-31). If we call *rand* and then use modulus 8, our range will be 0 through 7. To convert to the correct range, we first determine our modulus factor by subtracting the starting point (3) from the modulus divisor (8) and then adding the starting point to the resulting number. Thus, for our example, we subtract 3 from 8, which makes the modulus divisor 5.

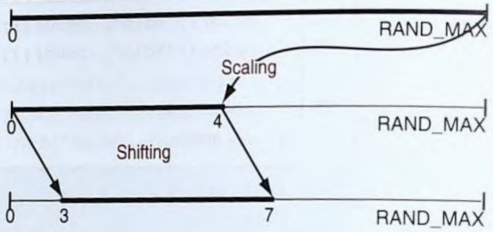


FIGURE 4-31 Random Number Scaling for 3–7

Generalizing the algorithm, we get

```
rand () % range + minimum
```

where range is (maximum – minimum) + 1, minimum is the minimum number and maximum is the maximum number in the desired range. For example, to create a random number in the range 10–20, we would use the following expression:

```
range  = (20 - 10) + 1;                        // 11
randNo = rand() % range + 10;
```

Program 4-11 generates a random number series in the range 10 to 20.

PROGRAM 4-11 Generating Random Numbers in the Range 10 to 20

```
/* Generate a random series in the range 10 to 20.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main (void)
```

continued

PROGRAM 4-11 Generating Random Numbers in the Range 10 to 20 *(continued)*

```
{
// Local Declarations
int range;

// Statements
   srand(time(NULL));
   range = (20 - 10) + 1;

   printf("%d",    rand() % range + 10);
   printf(" %d",   rand() % range + 10);
   printf(" %d\n", rand() % range + 10);

   return 0;
}  // main
```

```
Results:
10 11 16
```

Generating a Real Random Number Series

To generate a real random number series between 0.0 and 1.0 we divide the number returned by the random number generator by the maximum random number (RAND_MAX), which is found in *stdlib.h*, and then store it in a real number type. Program Program 4-12 demonstrates the generation of a real random number series in the range of 0 to 1.

PROGRAM 4-12 Generating Random Real Numbers

```
/* Generate a real random series in the range 0 to 1.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main (void)
{
// Local Declarations
   float x;

// Statements
   srand(time(NULL));

```

continued

PROGRAM 4-12 Generating Random Real Numbers *(continued)*

```
    x = (float)rand() / RAND_MAX;
    printf("%f", x);
    x = (float)rand() / RAND_MAX;
    printf(" %f", x);
    x = (float)rand() / RAND_MAX;
    printf(" %f\n", x);

    return 0;
}   // main
```

```
Results:
0.782006 0.264260 0.348460
```

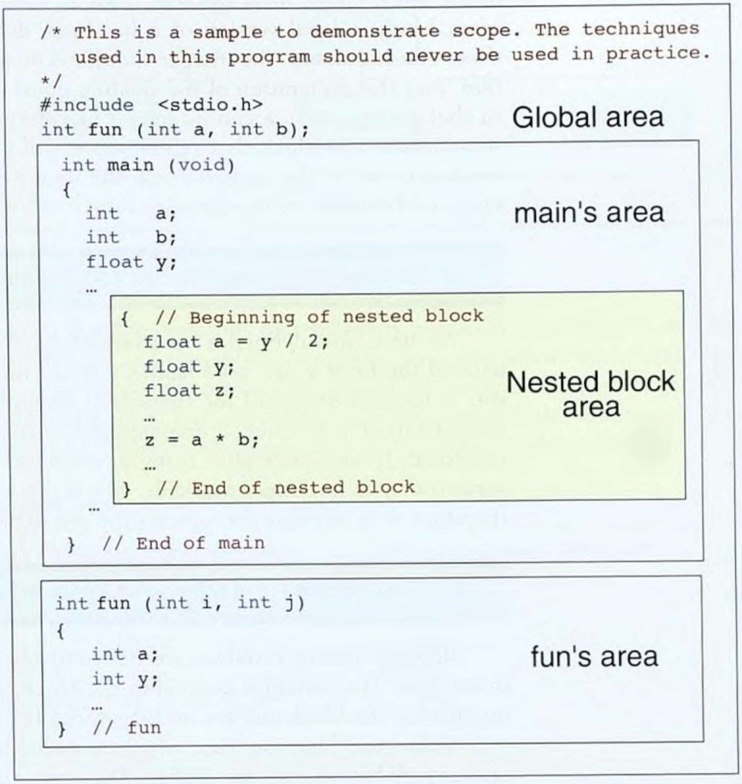
To scale a random real number into a range other than 0.0 to 1.0 (exclusive), we first generate the random number as we did in Program 4-12. Then we re-scale it to the new range. For example, to generate random real numbers in the range 100.0 to 300.0 (exclusive), we multiply the random number by the range (200.0) and then add the minimum (100.0). Using the results from Program 4-12, our random real numbers would be 256.4012, 152.852, and 169.692.

4.6 Scope

Scope determines the region of the program in which a defined object is *visible*—that is, the part of the program in which we can use the object's name. Scope pertains to any object that can be declared, such as a variable or a function declaration. It does not pertain directly to precompiler directives, such as define statements; they have separate rules. Scope is a source program concept: It has no direct bearing on the run-time program.

To discuss the concept of scope, we need to review two concepts. First, a block is zero or more statements enclosed in a set of braces. Recall that a function's body is enclosed in a set of braces; thus, a body is also a block. A block has a declarations section and a statement section. This concept gives us the ability to nest blocks within the body of a function and allows each block to be an independent group of statements with its own isolated definitions. Second, the global area of our program consists of all statements that are outside functions. Figure 4-32 is a graphical representation of the concept of global area and blocks.

An object's scope extends from its declaration until the end of its block. A variable is in scope if it is visible to the statement being examined. Variables are in scope from their point of declaration until the end of their block.



```
/* This is a sample to demonstrate scope. The techniques
   used in this program should never be used in practice.
*/
#include  <stdio.h>
int fun (int a, int b);
 int main (void)
 {
   int   a;
   int   b;
   float y;
   …
      {  // Beginning of nested block
        float a = y / 2;
        float y;
        float z;
        …
        z = a * b;
        …
      }  // End of nested block
   …
 }  // End of main
 int fun (int i, int j)
 {
   int a;
   int y;
   …
 }  // fun
```

FIGURE 4-32 Scope for Global and Block Areas

Global Scope

The global scope is easily defined. Any object defined in the global area of a program is visible from its definition until the end of the program. Referring to Figure 4-32, the function declaration for fun is a global definition. It is visible everywhere in the program.

Local Scope

Variables defined within a block have local scope. They exist only from the point of their declaration until the end of the block (usually a function) in which they are declared. Outside the block they are invisible.

In Figure 4-32, we see two blocks in *main*. The first block is all of *main*. Since the second block is nested within *main*, all definitions in *main* are visible to the block unless local variables with an identical name are defined. In the inner block, a local version of a has been defined; its type is float. Under these circumstances, the integer variable a in *main* is visible from its declaration until the declaration of the floating-point variable a in the nested block. At that point, *main*'s a can no longer be referenced in the nested block. Any statement in the block that references a will get the float version. Once we reach the end of the nested block, the float a is no longer in scope and the integer a becomes visible again.

Variables are in scope from declaration until the end of their block.

We have also defined a new variable y. Note, however, that before we defined the local y, we used *main*'s y to set the initial value for a. Although this is flagrant disregard for structured programming principles and should never be used in practice, it demonstrates that a variable is in scope until it is redefined. Immediately after using y, we defined the local version, so *main*'s version of y is no longer available. Since the variable b is not redeclared in the block, it is in scope throughout the entire block.

It is poor programming style to reuse identifiers within the same scope.

Although *main*'s variables are visible inside the nested block, the reverse is not true. The variables defined in the block, a, y, and z, exist only for the duration of the block and are no longer visible after the end of the block.

Within the function fun, which is coded after *main*, only its variables and any global objects are visible. Thus, we are free to use any names we want. In Figure 4-32, we chose to use the names a and y even though they had been used in *main*. This is an acceptable practice; there is nothing wrong with it.

4.7 Programming Example—Incremental Development

In Section 4.8, "Software Engineering," we discuss a concept known as "Top-Down Development." Top–down development, a concept inherent to modular programming, allows us to develop programs incrementally. By writing and debugging each function separately, we are able to solve the program in smaller steps, making the whole process easier. To demonstrate the concept, we begin the development of a calculator program in this section. In later chapters we add functionality.

The calculator program asks the user to input two numbers. The program then calls a function that adds the numbers. It concludes by displaying the sum of the two numbers.

The design is shown in Figure 4-33. It consists of a calculator function (*main*), and three subfunctions, getData, add, and printRes.

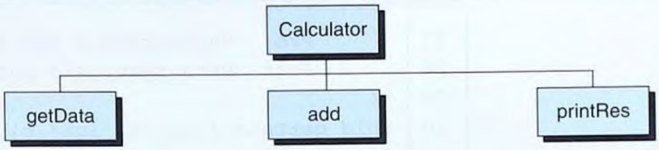


FIGURE 4-33 Calculator Program Design

First Increment: *main* and getData

When we develop programs incrementally, we begin by writing an abbreviated main function and one subfunction. Our first program also contains the necessary *include* statements and any global declarations that are required. It is shown in Program 4-13. Note that the program includes documentary comments as we understand them at this point in the development.

PROGRAM 4-13 Calculator Program—First Increment

```
/* This program adds two integers read from the
   keyboard and prints the results.
      Written by:
      Date:
*/
#include <stdio.h>

//  Function Declarations
void getData  (int* a, int* b);

int main (void)
{
// Local Declarations
   int  a;
   int  b;

// Statements
   getData (&a, &b);

   printf("**main:    a = %d; b = %d\n", a, b);

   return 0;
}  // main

/* ==================== getData ====================
   This function reads two integers from the keyboard.
```

continued

PROGRAM 4-13 Calculator Program—First Increment *(continued)*

```
       Pre   Parameters a and b are addresses
       Post  Data read into parameter addresses
*/
void getData (int* a, int* b)
{
   printf("Please enter two integer numbers: ");
   scanf("%d %d", a, b);

   printf("**getData: a = %d; b = %d\n", *a, *b);
   return;
} // getData
```

```
Results:
Please enter two integer numbers: 8 13
**getData: a = 8; b = 13
**main:    a = 8; b = 13
```

Program 4-13 Analysis It is very rare that we don't make some syntactical errors when writing even the simplest programs. By compiling the program in its simplest state, we are able to easily correct minor errors. We have also verified that the communication between functions is valid; that is, we verify that the variable types match the parameter types and that the data are being properly passed down and back up.

In this first incremental program, we pass the addresses of the two input variables to the get data function. It then reads and prints the input to verify that the data are correctly read. When we are back in *main*, we again print the input data to verify that it was properly passed up to *main*. Because we were careful to align the print formatting, we can easily check that the values are the same.

One point to note in the program is that we prefix temporary print statements with two asterisks. This allows us to tell the difference between permanent output and debugging output. As we move to the next increment, we remove the temporary statements from the debugged functions.

Second Increment: add

Once the first compile has been run and verified, we are ready to write the next function. While it is sometimes better to work in a different order, normally we select the next function in the program flow as shown in the structure chart, in this case the add function. This incremental version of the program is seen in Program 4-14.

PROGRAM 4-14 Calculator Program—Second Increment

```
/* This program adds two integers read
      from the keyboard and prints the results.
```

continued

PROGRAM 4-14 Calculator Program—Second Increment *(continued)*

```
      Written by:
      Date:
*/
#include <stdio.h>

//  Function Declarations
void getData  (int* a, int* b);
int  add      (int  a, int  b);

int main (void)
{
// Local Declarations
   int  a;
   int  b;
   int  sum;

// Statements
   getData (&a, &b);

   sum  = add  (a, b);
   printf("**main: %d + %d = %d\n", a, b, sum);
   return 0;
}  // main

/* ====================== getData ===================
   This function reads two integers from the keyboard.
      Pre   Parameters a and b are addresses
      Post  Data read into parameter addresses
*/
void getData (int* a, int* b)
{
   printf("Please enter two integer numbers: ");
   scanf("%d %d", a, b);
   return;
}  // getData

/* ===================== add =====================
   This function adds two numbers and returns the sum.
      Pre   a and b contain values to be added
      Post  Returns a + b
*/
int add (int a, int b)
{
// Local Definitions
```

continued

PROGRAM 4-14 Calculator Program—Second Increment *(continued)*

```
   int sum;

// Statements
   sum = a + b;
   printf("**add:  %d + %d = %d\n", a, b, sum);

   return sum;
}  // add
```

```
Results:
Please enter two integer numbers: 8 13
**add:  8 + 13 = 21
**main: 8 + 13 = 21
```

Program 4-14 Analysis Once again, we include print statements to verify that the add function received the correct data and properly returned it to *main*.

Final Increment: Print Results

The third and final increment includes the print results function. We do not need any debugging statements, however, because the print results function prints everything as a part of its requirements. The final program is shown in Program 4-15.

PROGRAM 4-15 Calculator Program—Final Increment

```
/* This program adds two integers read
   from the keyboard and prints the results.
      Written by:
      Date:
*/
#include <stdio.h>

//  Function Declarations
void getData  (int* a, int* b);
int  add      (int  a, int  b);
void printRes (int  a, int  b, int sum);

int main (void)
{
// Local Declarations
   int  a;
   int  b;
```

continued

PROGRAM 4-15 Calculator Program—Final Increment *(continued)*

```
   int  sum  = 0;

// Statements
   getData (&a, &b);

   sum  = add  (a, b);

   printRes (a, b, sum);
   return 0;
}  // main

/* ===================== getData ===================
   This function reads two integers from the keyboard.
      Pre   Parameters a and b
      Post  Returns a + b
*/
void getData (int* a, int* b)
{
   printf("Please enter two integer numbers: ");
   scanf("%d %d", a, b);
   return;
}  // getData

/* ===================== add =====================
   This function adds two integers and returns the sum.
      Pre   Parameters a and b
      Post  Returns a + b
*/
int add (int a, int b)
{
// Local Definitions
   int sum;

// Statements
   sum = a + b;
   return sum;
}  // add

/* =================== printRes ====================
   Prints the calculated results.
      Pre   a and b contain input; sum the results
      Post  Data printed
*/
```

continued

PROGRAM 4-15 Calculator Program—Final Increment *(continued)*

```
void printRes (int a, int b, int sum)
{
   printf("%4d + %4d = %4d\n", a, b, sum);
   return;
}  // printRes
```

```
Results:
Please enter two integer numbers: 8 13
   8 +   13 =   21
```

4.8 Software Engineering

In this section we discuss three different but related aspects of software engineering design: the structure chart, functional cohesion, and top–down development.

Structure Charts

The structure chart is the primary design tool for a program. Therefore, you should create it before you start writing your program. An analogy will help you understand the importance of designing before you start coding.

Assume that you have decided to build a house. You will spend a lot of time thinking about exactly what you want. How many rooms will it need? Do you want a family room or a great room? Should the laundry be inside the house or in the garage? To make sure everyone understands what you want, you will prepare formal blueprints that describe everything in detail. Even if you are building something small, like a dollhouse for a child or a toolshed for your back yard, you will make some sketches or plans.

Figuring out what you want in your house is comparable to determining the requirements for a large system. Drawing up a set of blueprints parallels the structure chart in the design of a program. All require advance planning; only the level of detail changes.

Professional programmers use the structure chart for another purpose. In a project team environment, before programmers start writing a program, they must have its design reviewed. This review process is called a structured walk-through. The review team consists of the systems analyst responsible for the area of the project, a representative of the user community, a system test engineer, and one or two programmers from the project.

The design walk-through serves three purposes: First, it ensures that you understand how your program fits into the system by communicating your design to the team. If the design has any omissions or communication errors, the team should detect them now. If you invite programmers who must interface with your program, you will also ensure that the interprogram communication linkages are correct.

Second, the walk-through validates your design. In creating your design, you will have considered several alternative approaches to writing your program. The review team will expect to see and understand the different designs you considered and hear why you chose the design you are proposing. They will challenge aspects of the design and suggest approaches you may not have considered. The result of the review will be the best possible design.

Finally, the walk-through gives the test engineer the opportunity to assess your program's testability. This in turn ensures that the final program will be robust and as error-free as possible.

Structure Chart Rules and Symbols

Figure 4-34 shows the various symbols that we use to write a structure chart. Although we include all symbols here for completeness, we will discuss only the two colored symbols. We discuss the other symbols in Chapters 5 and 6. In addition to the symbols, we will discuss several rules that we follow in designing our structure chart.

Function Symbol

Each rectangle in a structure chart (see Figure 4-34) represents a function *that we write.* Functions found in the standard C libraries are not shown. The name in the rectangle is the name we will give to the function when we write the program. It should be meaningful. The software engineering principle known as *intelligent names* states that the names used in a program should be self-documenting; that is, they should convey their intended usage to the reader. In general, we use intelligent names for both functions and for data names within our program.

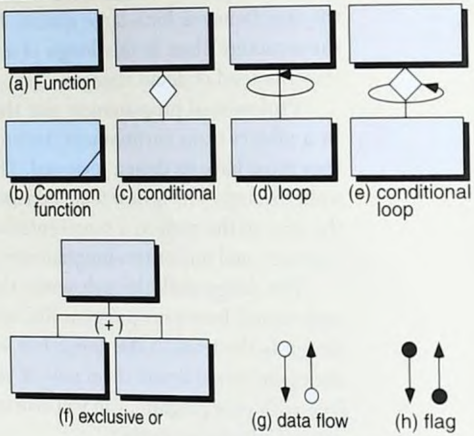


FIGURE 4-34 Structure Chart Symbols

Now that we have explained that all names should be descriptive, we will break our own rule because we want to concentrate on the format of a structure chart rather than a particular program. The names in Figure 4-35 identify the various modules for discussion.

Reading Structure Charts

Structure charts are read *top–down, left–right.* In Figure 4-35, ProgramName (*main*) consists of three subfunctions: `initialize`, `process`, and `endOfJob`. According to the left–right rule, the first call in the program is to `initialize`.

After initialize is complete, the program calls process. When process is complete, the program calls endOfJob. The functions on the same level of a structure chart are called in order from the left to the right.

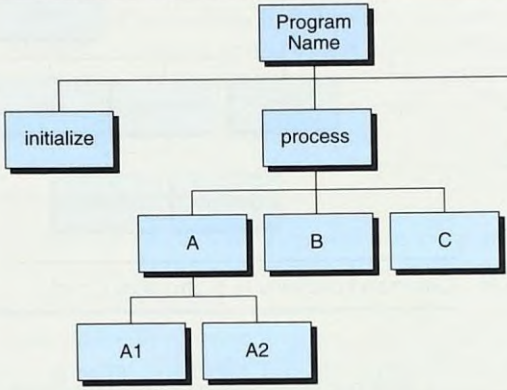


FIGURE 4-35 Structure Chart Design

The concept of top–down is demonstrated by process. When process is called, it calls A, B, and C in turn. Function B does not start running, however, until A is finished. While A is running, it calls A1 and A2 in turn. In other words, all functions in a line from process to A2 must be called before Function B can start.

No code is contained in a structure chart. A structure chart shows only the function flow through the program. It is not a block diagram or a flowchart. As a map of our program, the structure chart shows only the logical flow of the functions. Exactly how each function does its job is shown by algorithm design (flowchart or pseudocode). A structure chart shows the big picture; the details are left to algorithm design.

Structure charts show only function flow; they contain no code.

Often a program contains several calls to a common function (see Figure 4-34b). These calls are usually scattered throughout the program. The structure chart shows the call wherever it logically occurs in the program. To identify common structures, the lower right corner of the rectangle contains a cross-hatch or is shaded. If the common function is complex and contains subfunctions, these subfunctions must be shown only once. An indication that the incomplete references contain additional structure should be shown. This is usually done with a line below the function rectangle and a cut (~) symbol. This concept is shown in Figure 4-36, which uses a common function, average, in two different places in the program. Note, however, that we never graphically show a function connected to two calling functions.

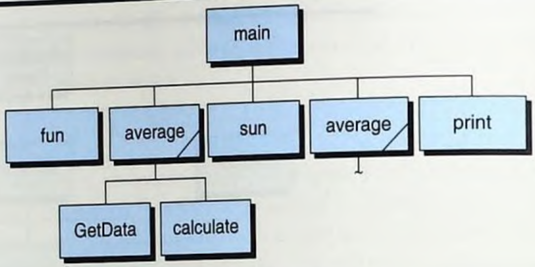


FIGURE 4-36 Common Functions in a Structure Chart

It is not necessary to show **data flows** (Figure 4-34g) and **flags** (see Figure 4-34h), both of which represent parameters, although it may be helpful in certain circumstances. If they are shown, inputs are on the left of the vertical line and outputs are on the right. When they are included, the name of the data or flag should also be indicated.

The structure chart rules described in this section are summarized in Table 4-1.

1. Each rectangle in a structure chart represents a function written by the programmer. Standard C functions are not included.
2. The name in the rectangle is an intelligent name that communicates the purpose of the function. It is the name that will be used in the coding of the function.
3. The function chart contains only function flow. No code is indicated.
4. Common functions are indicated by a cross-hatch or shading in the lower right corner of the function rectangle.
5. Common calls are shown in a structure wherever they will be found in the program. If they contain subfunction calls, the complete structure need be shown only once.
6. Data flows and flags are optional. When used, they should be named.
7. Input flows and flags are shown on the left of the vertical line; output flows and flags are shown on the right.

TABLE 4-1 Structure Chart Rules

Functional Cohesion

One of the most difficult structured programming concepts for new programmers is knowing when and how to create a function.

Functional cohesion is a measure of how closely the processes in a function are related. A function that does one and only one process is functionally cohesive. A function that contains totally unrelated processes is coincidentally cohesive. We provide a few rules here to help you write cohesive functions. For a complete discussion of the topic, see Page-Jones.[3]

Before we discuss the rules, however, you should understand why the concept is important. Following are the three primary reasons for using structurally cohesive functions:

1. *Correctness*: By concentrating on only one thing as you write a function, you will be less apt to make an error. It is much easier to get a simple task right than a complex task.
2. *Maintainability*: Production programs can live for years. The better structured a program, the easier it is to change. When programs are not well structured, making a change in one part of the program often leads to errors in other parts.
3. *Reusability*: Some processes are so common that they are found in many programs. Good programmers build libraries of these functions so that they don't have to reinvent the function each time it is needed. This not only leads to quicker program development but also reduces debugging time, since the library functions have already been debugged.

Only One Thing

Each function should do only one thing. Furthermore, all of the statements in the function should contribute only to that one thing. For example, assume that we are writing a program that requires the statistical measures of average and standard deviation. The two statistical measures are obviously related, if for no other reason than they are both measures of the same series of numbers. But we would not calculate both measures in one function. That would be calculating two things, and each function should do only one thing.

One way to determine if our function is doing more than one thing is to count the number of *objects* that it handles. An object in this sense is anything that exists separately from the other elements of the function. In the previous example, the average and the standard deviation are two different objects.

As another example, to compute the taxes for a payroll program in the state of California, we would deal with FICA taxes, state disability insurance, state unemployment taxes, state withholding taxes, and federal withholding taxes. Each of these is a different object. Our design should group all of these

[3] Meilir Page-Jones, *The Practical Guide to Structured Systems Design*, 2nd ed., Chap. 6. (Englewood Cliffs, N.J.: Prentice Hall, 1988).

taxes together in a function to calculate taxes, and it should call subfunctions to calculate each individual tax. This design is seen in Figure 4-37.

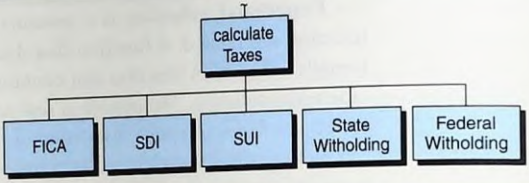


FIGURE 4-37 Calculate Taxes Design

In One Place

The corollary rule is that the one thing a function does should be done in only one place. If the code for a process is scattered in several different and unrelated parts of the program, it is very difficult to change. Therefore, all the processing for a task should be placed in one function and, if necessary, its subfunctions. This is the reason we created the function `calculateTaxes` in Figure 4-37.

An example of scattered code common among programmers is found in printing reports. Suppose that we needed to write a program that, among other things, prints a report that includes a heading, some data with a total, and then an end-of-report message. A well-structured solution is seen in Figure 4-38. It is quite common, however, to find the statements for each of these subtasks scattered in *main* and other parts of the program.

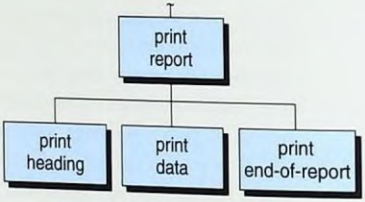


FIGURE 4-38 Design For Print Report

Testability

As a final measure of a program's structure, we should be able to test its functions independently. We will discuss a technique for this in the next section. Let us simply say here that **testability** is an attribute of a well-designed and well-structured program that allows each section of the program to be tested separately from the rest of the program.

Top–Down Development

If we have designed our program using structured programming concepts and a structure chart, we can then proceed to implement it in a top–down fashion.

Referring again to Figure 4-35, "Structure Chart Design," a top–down implementation starts with the code for *main* only, shown in the structure chart as `ProgramName`. The code for the first compile and test is shown in Program 4-16.

PROGRAM 4-16 Top–down Development Example

```
/* Sample of top-down development using stubs.
      Written by:
      Date:
*/
#include <stdio.h>

//  Function Declarations
   int initialize (void);
   int process    (void);
   int endOfJob   (void);

int main (void)
{
// Statements
   printf("Begin program \n\n");

   initialize ();
   process ();
   endOfJob ();
   return 0;
}  // main

/* ================= initialize ===================
   Stub for initialize.
*/
int initialize (void)
{
   // Statements
   printf("In initialize: \n");
   return 0;
}  // initialize
/* =================== process ===================
   Stub for process
*/
int process (void)
```

continued

PROGRAM 4-16 Top–down Development Example *(continued)*

```
{
   // Statements
   printf("In process: \n");
   return 0 ;
}  // process
/* ================== endOfJob ==================
   Stub for endOfJob
*/
int endOfJob (void)
{
   // Statements
   printf("In endOfJob: \n");
   return 0 ;
}  // endOfJob
```

```
Results:
Begin program
In initialize:
In process:
In endOfJob:
```

Program 4-16 Analysis Note that this program has only the first four boxes from the structure chart: `ProgramName` (*main*), `initialize`, `process`, and `endOfJob`. For each of *main*'s subfunctions, all that is included is a **stub**. A stub is the skeleton of a function that is called and immediately returns. Although it is a complete function, it does nothing other than to establish and verify the linkage between the caller and itself. But this is a very important part of testing and verifying a program. At this point the program should be compiled, linked, and run. Chances are that you will find some minor problems, such as missing semicolons or errors between the function declarations and the function definitions. Before you continue with the program, you should correct these problems.

The top–down development then continues with the coding of `initialize`, `process`, or `endOfJob`. Normally, you develop the functions left to right, but it is not necessary to do so. To develop `process`, you again stub its subfunctions, `A`, `B`, and `C`, and then test the program. This top–down development continues until the complete program has been coded and tested.

4.9 Tips and Common Programming Errors

1. Several possible errors are related to passing parameters.

   a. It is a compile error if the types in the function declaration and function definition are incompatible. For example, the types in the following statements are incompatible:

```
double divide (int dividend, int divisor);
       …
double divide (float dividend, float divisor)
{
       …
}      // divide
```

   b. It is a compile error to have a different number of actual parameters in the function call than there are in the function declaration.

   c. It is a logic error if you code the parameters in the wrong order. Their meaning will be inconsistent in the called program. For example, in the following statements, the types are the same but the meaning of the variables is reversed:

```
double divide (float dividend, float divisor);
       …
double divide (float divisor,  float dividend)
{
       …
}      // divide
```

2. It is a compile error to define local variables with the same identifiers as formal parameters, as shown below.

```
double divide (float dividend, float divisor)
{
//     Local Declarations
       float dividend;
       …
}      // divide
```

3. Using a *void* return with a function that expects a return value or using a return value with a function that expects a *void* return is a compile error.

4. Each parameter's type must be individually specified; you cannot use multiple definitions like you can in variables. For example, the following is a compile error because y does not have a type:

```
double fun (float x, y);
```

5. Forgetting the semicolon at the end of a function declaration is a compile error. Similarly, using a semicolon at the end of the header in a function definition is a compile error.
6. It is most likely a logic error to call a function from within itself or one of its called functions. (This is known as recursion, and its correct use is covered in Chapter 6.)
7. It is a compile error to attempt to define a function within the body of another function.
8. It is a run-time error to code a function call without the parentheses, even when the function has no parameters.

```
printHello;                              // Not a call
printHello ( );                          // A valid call
```

9. It is a compile error if the type of data in the return statement does not match the function return type.
10. It is a logic error to call *srand* every time you call *rand*.

4.10 Key Terms

absolute value
actual parameters
bi-directional communication
called function
calling function
ceiling
data flow
decomposition
dereferencing
downward communication
flag
floor
formal parameter list
function body
function call
functional cohesion
function declaration
function definition
function header
indirection operator
local variables
parameter passing
pass by reference
pass by value
pseudorandom number series
random number series
return
scaling (with random numbers)
scope
random number seed
stub
testability
top–down design
upward communication

4.11 Summary

- ❑ In structured programming, a program is divided into modules.
    - ■ Each module is designed to do a specific task.
    - ■ Modules in C are written as functions.

- Each C program must have one and only one function called *main*.
- A function can return only one value.
- A function can be called for its returned value or for its side effect.
- The function call includes the function name and the values of the actual parameters to provide the called function with the data it needs to perform its job.
- Each actual parameter of the function is an expression. The expression must have a value that can be evaluated at the time the function is called.
- A local variable is known only in a function definition. The local variables do not take part in communication between the calling and the called functions.
- The general format for a function definition is

```
return_type function_name (parameter list)
{
// Local Declarations

// Statements
}  // function_name
```

- If a function returns no value, the return type must be declared as *void*.
- If a function has no parameters, the parameter list must be declared *void*.
- The actual parameters passed to a function must match in number, type, and order with the formal parameters in the function definition.
- When a function is called, control is passed to the called function. The calling function "rests" until the called function finishes its job.
- We highly recommend that every function have a *return* statement. A *return* statement is required if the return type is anything other than *void*.
- Control returns to the caller when the *return* statement is encountered.
- A function declaration requires only the return type of the function, the function name, and the number, types, and order of the formal parameters. Parameter identifiers may be added for documentation but are not required.
- The scope of a parameter is the block following the header.
- A local variable is a variable declared inside a block. The scope of a local variable is the block in which it is declared.
- Structure charts are used to design a program.
- Structure charts contain no code.
- Functional cohesion is a measure of how closely the processes in a function are related.
- Programs should be written using a top–down implementation with stubs for functions not yet coded.

4.12 Practice Sets

Review Questions

1. The principles of top–down design and structured programming dictate that a program should be divided into a main module and its related modules.
   - a. True
   - b. False
2. The function definition contains the code for a function.
   - a. True
   - b. False
3. Function calls that return *void* may not be used as a part of an expression.
   - a. True
   - b. False
4. The address operator (`&`) is used to tell the compiler to store data at an address.
   - a. True
   - b. False
5. Variables defined within a block have global scope.
   - a. True
   - b. False
6. The process of dividing a program into functions—which in turn are divided into functions until they consist of only elementary processing that is intrinsically understood and cannot be further subdivided—is known as
   - a. charting
   - b. flow charting
   - c. factoring
   - d. programming
   - e. structuring
7. Which of the following statements about function declaration and definition is true?
   - a. The function call is found in the called function.
   - b. The function declaration requires that the parameters be named.
   - c. The function definition is done with a function declaration.
   - d. The function definition contains executable statements that perform the function's task.
   - e. The function definition header concludes with a semicolon (`;`).

8. Which of the following is not a part of a function header?
   - a. name
   - b. parameter list
   - c. return type
   - d. title

9. Which of the following statements about function parameters is true?
   - a. Empty parameter lists are declared with the keyword *void*.
   - b. If there is only one parameter, the function list parentheses are not required.
   - c. In the definition of a function, the parameters are known as actual parameters.
   - d. Parameters are separated by semicolons.
   - e. The parameters in a function definition are defined in the function's body (local declaration section).

10. Which of the following statements about local variables is false?
    - a. A local variable's value may be returned through a *return* statement.
    - b. Local variables are defined inside a function.
    - c. Local variables cannot be referenced through their identifiers outside the function.
    - d. Local variables may be initialized with an initializer.
    - e. Local variables' names can be the same as the function's parameter names.

11. To tell the compiler to store data at an address, use the
    - a. address operator (`&`)
    - b. array operator (`[ ]`)
    - c. dereference operator (`#`)
    - d. indirection operator (`*`)
    - e. pointer operator (`^`)

12. The function that returns the absolute value of a long integer is
    - a. `abs`
    - b. `dabs`
    - c. `fabs`
    - d. `labs`
    - e. `tabs`

13. Which of the following statements will generate a random number in the range 30–50?
    - a. `rand (33)`
    - b. `(rand () % 20) + 1`
    - c. `(rand () % 21) + 20`
    - d. `(rand () % 21) + 30`
    - e. `(rand () % 51) + 1`

14. Which of the following statements about structure charts is false?
    a. Structure charts are a replacement for flowcharts.
    b. Structure charts are the primary design tool for a program.
    c. Structure charts are used in a structured walk-through to validate the design.
    d. Structure charts can be used to assess the testability of a program.
    e. Structure charts should be created before you start writing a program.

Exercises

15. Find any errors in the following function definition:

```
void fun (int x, int y)
{
 int z;
 …
 return z;
} // fun
```

16. Find any errors in the following function definition:

```
int fun (int x, y)
{
 int z;
 …
 return z;
} // fun
```

17. Find any errors in the following function definition:

```
int fun (int x, int y)
{
 …
 int sun (int t)
 …
     {
      …
      return (t + 3);
     }
 …
 return z;
} // fun
```

18. Find any errors in the following function definition:

```
void fun (int, x)
{
```

continued

```
         …
         return;
        } // fun
```

19. Find any errors in the following function declarations:

    a. `int sun (int x, y);`
    b. `int sun (int x, int y)`
    c. `void sun (void, void);`
    d. `void sun (x int, y float);`

20. Find any errors in the following function calls:

    a. `void fun ( );`
    b. `fun (void);`
    c. `void fun (int x, int y);`
    d. `fun ( );`

21. Evaluate the value of the following expressions:

    a. `fabs (9.5)`
    b. `fabs (-2.4)`
    c. `fabs (-3.4)`
    d. `fabs (-7)`
    e. `fabs (7)`

22. Evaluate the value of the following expressions:

    a. floor (9.5)
    b. `floor ( -2.4 )`
    c. `floor ( -3.4 )`
    d. `ceil ( 9.5 )`
    e. `ceil ( -2.4 )`
    f. `ceil ( -3.4 )`

23. Evaluate the value of the following expressions when x is 3.5, 3.45, 3.76, 3.234, and 3.4567:

    a. `floor (x * 10 + 0.5) / 10`
    b. `floor (x * 100 + 0.5) / 100`
    c. `floor (x * 1000 + 0.5) / 1000`

24. Define the range of the random numbers generated by the following expressions:

    a. `rand( )  % 10`
    b. `rand( )  % 4`
    c. `rand( )  % 10 + 1`
    d. `rand( )  % 52`
    e. `rand( )  % 2 + 1`
    f. `rand( ) % 52 - 5`

25. What would be printed from Program 4-17 when run using `3   5` as data?

PROGRAM 4-17 Program for Exercise 25

```
#include <stdio.h>

// Function Declarations
int strange (int x, int y);

int main (void)
{
// Local Declarations
    int  a;
    int  b;
    int  r;
    int  s;

// Statements
   scanf("%d %d", &a, &b);
   r = strange (a, b);
   s = strange (b, a);
   printf("%d %d", r, s);
   return 0;
}  // main
// ================= strange =================
int strange (int x, int y)
{
// Statements
   return (x - y);
}  // strange
```

26. What would be printed from Program 4-18 when run using 3 5 4 6 as data?

PROGRAM 4-18 Program for Exercise 26

```
#include <stdio.h>

// Function Declarations
int strange (int x, int y);

int main (void)
{
// Local Declarations
    int  a;
    int  b;
    int  c;
    int  d;
    int  r;
```

continued

PROGRAM 4-18 Program for Exercise 26 *(continued)*

```
    int  s;
    int  t;
    int  u;
    int  v;

// Statements
   scanf("%d %d %d %d", &a, &b, &c, &d);

   r = strange (a, b);
   s = strange (r, c);
   t = strange (strange (s, d), strange (4, 2));
   u = strange (t + 3, s + 2);
   v = strange (strange (strange (u, a), b), c);

   printf("%d %d %d %d %d", r, s, t, u, v);
   return 0;
}  // main
// ================== strange =================
int strange (int x, int y)
{
// Local Declarations
    int  t;
    int  z;

// Statements
    t = x + y;
    z = x * y;
    return (t + z);
}  // strange
```

27. Draw the structure chart for Program 4-19. What output does it produce?

PROGRAM 4-19 Program for Exercise 27

```
#include <stdio.h>

// Function Declarations
int  funA (int x);
void funB (int x);

int main (void)
{
// Local Declarations
   int  a;
```

continued

PROGRAM 4-19 Program for Exercise 27 *(continued)*

```
   int  b;
   int  c;

// Statements
   a = 10;
   funB (a);
   b = 5;
   c = funA (b);
   funB(c);
   printf("%3d %3d %3d", a, b, c);
   return 0;
}  // main

int funA (int x)
{
// Statements
   return x * x;
}  // funA

void funB (int x)
{
// Local Declarations
   int y;

// Statements
   y  = x % 2;
   x /= 2;
   printf ("\n%3d %3d\n", x, y);
   return;
}  // funB
```

Problems

28. Write a function to print your name, as shown below. Write a call as it would be coded in a calling function, such as *main*.

```
*****************************
*                           *
*       Your Name Here      *
*                           *
*****************************
```

29. Write a program that generates a random number from the following set:

```
1, 2, 3, 4, 5, 6
```

30. Write a program that generates a random number from the following set:

```
1, 4, 7, 10, 13, 16
```

31. Explain what is meant by the statement "a function should do only one thing."

32. Code and run Program 4-16, "Top–down Development Example," to demonstrate how stubs work.

33. Write a function to convert inches into centimeters. (One inch is 2.54 centimeters.) Then write a program that prompts the user to input a measure in inches, calls the conversion function, and prints out the measurement in centimeters.

34. Write a program that reads three integers and then prints them in the order read and reversed. Use four functions: *main*, one to read the data, one to print them in the order read, and one to print them reversed.

35. Expand the calculator program, Program 4-15, to calculate the difference, product, quotient, and modulus of the numbers. Calculate the quotient and modulus in one function.

36. Modify Program 4-5, "Add Two Digits," to add the least significant three digits (hundreds, tens, and ones).

37. Write a function that receives a positive floating-point number and rounds it to two decimal places. For example, 127.565031 rounds to 127.570000. Hint: To round, you must convert the floating-point number to an integer and then back to a floating-point number. Print the rounded numbers to six decimal places. Test the function with the following data:

```
123.456789  123.499999 123.500001
```

38. Write a program that reads a floating-point number and prints the ceiling, floor, and rounded value. Use the function in Problem 37 for the average and test it with the same values as Problem 37.

39. Write a function to compute the perimeter and area of a right triangle (Figure 4-39) when given the length of the two sides (a and b).

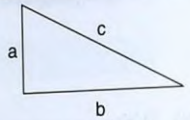


FIGURE 4-39 Problem 39

The following formulas may be helpful:

```
c² = a² + b²
area = .5 * (a * b)
```

Projects

40. Prepare a payroll earnings statement for the sales force at the Arctic Ice Company. All of Arctic's employees are on a straight commission basis of 12.5% of sales. Each month, they also receive a bonus that varies depending on the profit for the month and their length of service. The sales manager calculates the bonus separately and enters it with the salesperson's total sales for the month. Your program is also to calculate the withholding taxes and retirement for the month based on the following rates:

a. 25% Federal withholding

b. 10% State withholding

c. 8% Retirement plan

Use the test data in Table 4-2 to test the program.

Salesperson	Sales	Bonus
1	53,500	425
2	41,300	300
3	56,800	350
4	36,200	175

TABLE 4-2 Test Data for Project 40

41. Write a program that, given a beginning balance in your savings account, calculates the balance at the end of 1 year. The interest is 5.3% compounded quarterly. Show the interest earned and balance at the end of each quarter. Present the data in tabular columns with appropriate headings. Use separate functions to compute the interest and print the balance.

42. The formula for converting centigrade temperatures to fahrenheit is

$$F = 32 + C\frac{180.0}{100.0}$$

Write a program that asks the user to enter a temperature reading in centigrade and then prints the equivalent Fahrenheit value. It then asks the user to enter a Fahrenheit value and prints out the equivalent centigrade value. Run the program several times. Be sure to include at least one negative temperature reading in your test cases. Provide separate functions as needed by your design. One possible design is shown in Figure 4-40. (Your *main* function should have only function calls.)

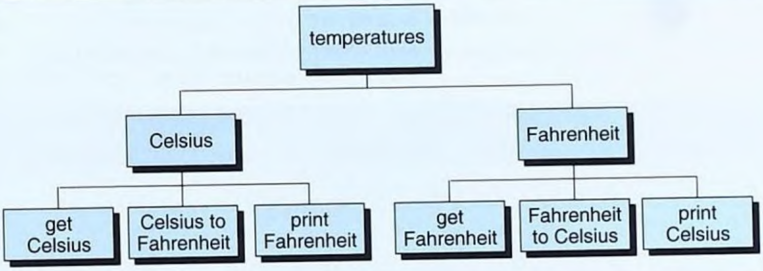


FIGURE 4-40 A Possible Design for Project 44

43. Write a program that uses standard functions. The program may be written entirely in *main* and must follow the pseudocode shown in Algorithm 4-1. Give the output appropriate captions, and align the data.

ALGORITHM 4-1 Pseudocode for Project 43

```
Prompt the user to enter a number
Read number
Display number
Get a random number and scale to range 3…37
Display random number
Set product to number * random number
Display product
Display ceiling of random number
Display floor of product
Display number raised to power of random number
Display square root of random number
```

44. Write a C program that creates customers' bills for a carpet company when the following information is given:

a. the length and the width of the carpet in feet

b. the carpet price per square foot

c. the percent of discount for each customer

The labor cost is fixed at $0.35 per square foot. It is to be defined as a constant. The tax rate is 8.5% applied after the discount. It is also to be defined as a constant. The input data consist of a set of three integers representing the length and width of the room to be carpeted, the percentage of the discount the owner gives to a customer, and a real number representing the unit price of the carpet. The program is to prompt the user for this input as shown below. (Colored numbers are typical responses.)

```
Length of room (feet)?          30
Width of room (feet)?           18
Customer discount (percent)?     9
Cost per square foot (xxx.xx)? 8.23
```

The output is shown below. Be careful to align the decimal points.

```
                    MEASUREMENT

      Length                            XXX ft
      Width                             XXX ft
      Area                              XXX square ft

                      CHARGES

      DESCRIPTION  COST/SQ.FT.      CHARGE
      -----------  -----------   -----------
      Carpet         XXX.XX       $XXXX.XX
      Labor            0.35        XXXX.XX
                                 -----------
      INSTALLED PRICE             $XXXX.XX
      Discount          XX%        XXXX.XX
                                 -----------
      SUBTOTAL                    $XXXX.XX
      Tax                          XXXX.XX
      TOTAL                       $XXXX.XX
```

The program's design should use *main* and at least the six functions described below:

a. Read data from the keyboard. This function is to use addresses to read all data and place them in the calling function's variables.

b. Calculate values. This function calls three subfunctions. Each function is to use addresses to store their results.

- □ Calculate the installed price. This function calculates area, carpet cost, labor cost, and installed price. The installed price is the cost of the carpet and the cost of the labor.
- □ Calculate the subtotal. This function calculates the discount and subtotal.
- □ Calculate the total price with discount and tax. This function calculates the tax and the total price.

c. Print the result. Use two subfunctions to print the results: one to print the measurements, and one to print the charges.

Test your program with the test data shown in Table 4-3.

Test	Length	Width	Discount	Cost
1	23	13	12	$14.20
2	35	8	0	$ 8.00
3	14	11	10	$22.25

TABLE 4-3 Test Data for Project 44

The program [illegible] should [illegible]

[illegible]

Read data from the [illegible]. The [illegible]

[illegible]

[illegible]

[illegible]
[illegible]

Chapter 5

Selection—Making Decisions

In Chapter 1's "Software Engineering" section, we mentioned that Dijkstra proposed that any program could be written with three constructs: sequence, selection, and loop. The C language implementation of the first construct, sequence, is covered in the previous chapters. In this chapter, we turn our attention to the second construct, selection. **Selection** allows us to choose between two or more alternatives: It allows us to make decisions.

What a dull world it would be if we didn't have any choices. Vanilla ice cream for everybody! Uniforms all around! And no debates or arguments to keep things interesting. Fortunately, our world is filled with choices. And since our programs must reflect the world in which they are designed to operate, they, too, are filled with choices and opportunities for decision making.

How are decisions made by a computer? In this chapter, you will find out. One of the main points to keep in mind is that the decisions made by a computer must be very simple, since everything in the computer ultimately reduces to either true or false. If complex decisions are required, it is the programmer's job to reduce them to a series of simple decisions that the computer can handle.

Objectives

□ To understand how decisions are made in a computer

□ To understand the logical operators: and, or, and not

□ To understand how a C program evaluates a logical expression

□ To write programs using logical and comparative operators

□ To write programs that use two-way selection: *if...else* statements

□ To write programs that use multi-way selection: *switch* and *else...if*

□ To understand C's classification of characters

□ To write programs that use C's character functions

□ To be able to design a structure chart that specifies function selection

5.1 Logical Data and Operators

A piece of data is called logical if it conveys the idea of true or false. We need logical data in real life as well as in programming. In real life, logical data (true or false) are created in answer to a question that needs a yes–no answer. For example, we ask if an item is on sale or not. We ask if a business is open or not. The answer to these questions is a piece of data that is usually yes or no. We can also ask questions such as "Is x greater than y?" The answer is again yes or no. In computer science, we do not use yes or no, we use true or false.

Logical Data in C

Traditionally, C had no logical data type; C programmers used other data types, such as *int*, to represent logical data. If a data value is zero, it is considered false. If it is nonzero, it is considered true. This concept of true and false on a numeric scale is seen in Figure 5-1.

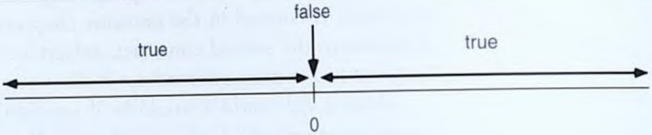


FIGURE 5-1 True and False on the Arithmetic Scale

The C99 standard introduced the Boolean data type, *_bool*, which is declared as an unsigned integer in the *stdbool.h* header file. Also defined in the header files are the identifiers *bool*, *true*, and *false*. While the standard still supports the traditional use of integers for logical data, we recommend that the Boolean type, *bool*, be used.

Logical Operators

C has three **logical operators** for combining logical values and creating new logical values: *not*, *and*, and *or*. A common way to show logical relationships is in truth tables. Truth tables list the values that each operand can assume and the resulting value. The truth tables for logical operators are shown in Figure 5-2.

not Operator

The *not* operator (`!`) is a unary operator with precedence 15 in the Precedence Table (see inside front cover). It changes a *true* value to *false* and a *false* value to *true*.

not

x	!x
false	true
true	false

and

x	y	x&&y
false	false	false
false	true	false
true	false	false
true	true	true

or

x	y	x||y
false	false	false
false	true	true
true	false	true
true	true	true

FIGURE 5-2 Logical Operators Truth Table

and Operator

The *and* operator (&&) is a binary operator with a precedence of 5. Since *and* is a binary operator, four distinct combinations of values in its operands are possible. The result is *true* only when both operands are *true*; it is *false* in all other cases.

or Operator

The *or* operator (||) is a binary operator with a precedence of 4. Again, since it is a binary operator, four distinct combinations of values in its operands are possible. The result is *false* if both operands are *false*; it is *true* in all other cases.

Evaluating Logical Expressions

Computer languages can use two methods to evaluate the binary logical relationships. In the first method, the expression must be completely evaluated before the result is determined. This means that the *and* expression must be completely evaluated, even when the first operand is *false* and it is therefore known that the result must be *false*. Likewise, in the *or* expression, the whole expression must be evaluated even when the first operand is *true* and the obvious result of the expression must be *true*.

The second method sets the resulting value as soon as it is known. It does not need to complete the evaluation. In other words, it operates in a short-circuit fashion and stops the evaluation when it knows for sure what the final result will be. Under this method, if the first operand of a logical *and* expression is *false*, the second half of the expression is not evaluated because it is apparent that the result must be *false*. Again, with the *or* expression, if the first operand is *true*, there is no need to evaluate the second half of the expression, so the resulting value is set *true* immediately. C uses this short-circuit method, which is graphically shown in Figure 5-3.

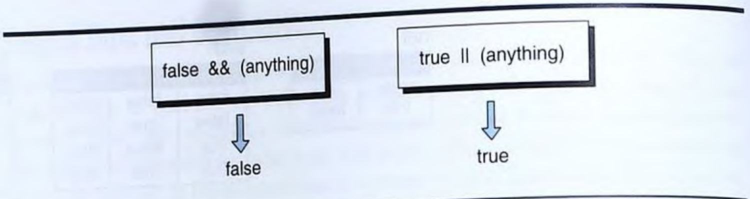


FIGURE 5-3 Short-circuit Methods for *and*/*or*

Although the C method is more efficient, it can cause problems when the second operand contains side effects, which is poor programming practice. Consider for example, the following expression in which a programmer wants to find the value of the logical expression and at the same time wants to increment the value of the second operand:

```
x && y++
```

Everything works fine when the first operand is *true*. But if the first operand is *false*, the second operand will never be evaluated and therefore will never be incremented. It is the same with the following *or* example. If the first operand is *true*, the second operand will never be incremented.

```
x || y++
```

Program 5-1 demonstrates the use of logical data in expressions.

PROGRAM 5-1 Logical Expressions

```
/* Demonstrate the results of logical operators.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdbool.h>

int main (void)
{
// Local Declarations
   bool a = true;
   bool b = true;
   bool c = false;

// Statements
```

continued

PROGRAM 5-1 Logical Expressions *(continued)*

```
    printf("    %2d AND     %2d: %2d\n", a, b,  a &&  b);
    printf("    %2d AND     %2d: %2d\n", a, c,  a &&  c);
    printf("    %2d AND     %2d: %2d\n", c, a,  c &&  a);
    printf("    %2d OR      %2d: %2d\n", a, c,  a ||  c);
    printf("    %2d OR      %2d: %2d\n", c, a,  c ||  a);
    printf("    %2d OR      %2d: %2d\n", c, c,  c ||  c);
    printf("NOT %2d AND NOT %2d: %2d\n", a, c, !a && !c);
    printf("NOT %2d AND     %2d: %2d\n", a, c, !a &&  c);
    printf("    %2d AND NOT %2d: %2d\n", a, c,  a && !c);
    return 0;
}   // main
```

```
Results:
     1 AND      1:  1
     1 AND      0:  0
     0 AND      1:  0
     1 OR       0:  1
     0 OR       1:  1
     0 OR       0:  0
NOT  1 AND NOT  0:  0
NOT  1 AND      0:  0
     1 AND NOT  0:  1
```

Program 5-1 Analysis Each print statement in Program 5-1 contains a logical expression that evaluates either to *true* or *false*. Make sure you understand why each of the expressions evaluates as shown in the results. Note that even though the results are of type Boolean, they must be printed as integer (%d). This is because while the expressions evaluate to *true* or *false*, there is no conversion code for Boolean. Therefore, when we print them, we must use the integer field specification.

Comparative Operators

In addition to logical operators, C provides six **comparative operators**. Comparative operators are divided into two categories, relational and equality. They are all binary operators that accept two operands and compare them. The result is a Boolean type—that is, the result is always *true* or *false*. The operators are shown in Figure 5-4.

The relational operators—less than, less than or equal, greater than, greater than or equal—have a higher priority (10) in the Precedence Table (see inside front cover) than the equality and not equal operators (9). This means that relational operators are evaluated before the equality operators when they appear together in the same expression.

It is important to recognize that each operator is a complement of another operator in the group. But, surprisingly, the complement is not the one that you might expect. Figure 5-5 shows each operator and its complement.

Type	Operator	Meaning	Precedence
Relational	<	less than	10
	<=	less than or equal	
	>	greater than	
	>=	greater than or equal	
Equality	==	equal	9
	!=	not equal	

FIGURE 5-4 Relational and Equality Operators

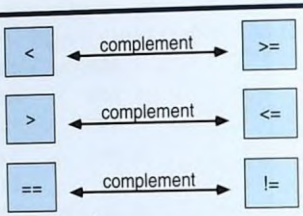


FIGURE 5-5 Comparative Operator Complements

If we want to simplify an expression involving the *not* and the *less than* operator, we use the *greater than* or *equal* operator. This concept is important for simplifying expressions and coding expressions in good, clear style. Table 5-1 shows an example of each expression and its simplified version.

Original Expression	Simplified Expression
`!(x <  y)`	`x >= y`
`!(x >  y)`	`x <= y`
`!(x != y)`	`x == y`
`!(x <= y)`	`x >  y`
`!(x >= y)`	`x <  y`
`!(x == y)`	`x != y`

TABLE 5-1 Examples of Simplifying Operator Complements

Program 5-2 demonstrates the use of the comparative operators.

PROGRAM 5-2 Comparative Operators

```c
/* Demonstrates the results of relational operators.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int a = 5;
   int b = -3;

// Statements *
   printf(" %2d <  %2d is %2d\n", a, b,  a <  b);
   printf(" %2d == %2d is %2d\n", a, b,  a == b);
   printf(" %2d != %2d is %2d\n", a, b,  a != b);
   printf(" %2d >  %2d is %2d\n", a, b,  a >  b);
   printf(" %2d <= %2d is %2d\n", a, b,  a <= b);
   printf(" %2d >= %2d is %2d\n", a, b,  a >= b);
   return 0;
}  // main
```

```
Results:
5 <  -3 is  0
5 == -3 is  0
5 != -3 is  1
5 >  -3 is  1
5 <= -3 is  0
5 >= -3 is  1
```

Program 5-2 Analysis Program 5-2 follows the same patterns we saw in Program 5-1. Once again, make sure that you understand why each of the representations in the results evaluates to *true* or *false*.

5.2 Two-Way Selection

The basic decision statement in the computer is the **two-way selection**. The decision is described to the computer as a conditional statement that can be answered either *true* or *false*. If the answer is *true*, one or more action statements are executed. If the answer is *false*, then a different action or set of actions is executed. Regardless of which set of actions is executed, the program continues with the next statement after the selection. The flowchart for two-way decision logic is shown in Figure 5-6.

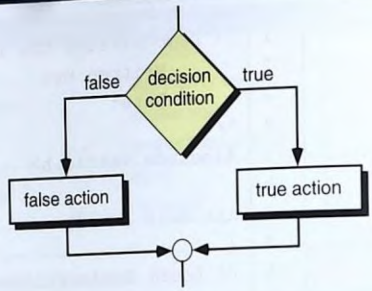
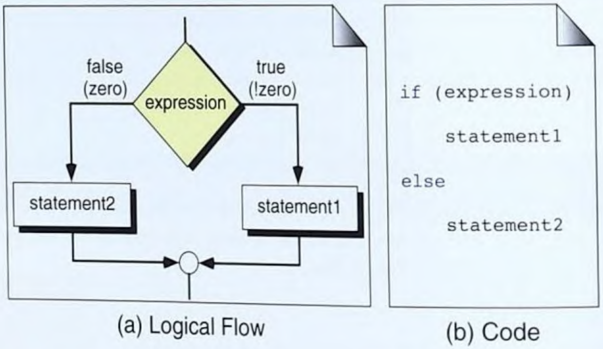


FIGURE 5-6 Two-way Decision Logic

if...else

C implements two-way selection with the *if...else* statement. An *if...else* statement is a composite statement used to make a decision between two alternatives. Figure 5-7 shows the logic flow for an *if...else* statement. The expression can be any C expression. After it has been evaluated, if its value is *true*, `statement1` is executed: otherwise, `statement2` is executed. It is impossible for both statements to be executed in the same evaluation.



```
if (expression)
    statement1
else
    statement2
```

(a) Logical Flow

(b) Code

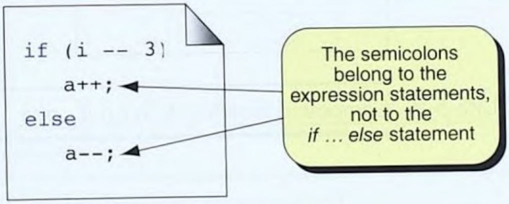
FIGURE 5-7 *if...else* Logic Flow

There are some syntactical points we must remember about *if...else* statements. These points are summarized in Table 5-2.

1. The expression must be enclosed in parentheses.
2. No semicolon (;) is needed for an *if...else* statement; statement 1 and statement 2 may have a semicolon as required by their types.
3. The expression can have a side effect.
4. Both the true and the false statements can be any statement (even another *if...else* statement) or they can be a null statement.
5. Both statement 1 and statement 2 must be one and only one statement. Remember, however, that multiple statements can be combined into a compound statement through the use of braces.
6. We can swap the position of statement 1 and statement 2 if we use the complement of the original expression.

TABLE 5-2 Syntactical Rules for *if...else* Statements

The first rule in Table 5-2—that the expression must be enclosed in parentheses—is simple and needs no further discussion. The second rule is also simple, but it tends to cause more problems. We have therefore provided an example in Figure 5-8. In this example, each action is a single statement that either adds or subtracts 1 from the variable a. Note that the semicolons belong to the arithmetic statements, not the *if...else* statement.



```
if (i -- 3)
    a++;
else
    a--;
```

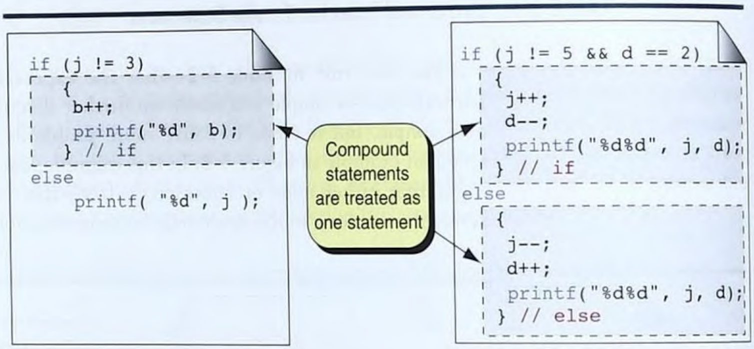
The semicolons belong to the expression statements, not to the *if ... else* statement

FIGURE 5-8 A Simple *if...else* Statement

The third rule in Table 5-2 requires more discussion. It is quite common in C to code expressions that have side effects. For example, we often use expressions that read data as a side effect. As an example, consider what happens when we are writing a series of numbers, and when we have written ten numbers we want to go to a new line. A simple solution increments a line count and tests the limit in the same statement. The code for this logic could be written as shown in the following example.

```
if (++lineCnt > 10)
       {
        printf("\n");
        lineCnt = 0;
       } // end true

printf( … );
```

Rules 4 and 5 in Table 5-2 are closely related. The fact that any statement can be used in an *if...else* is straightforward, but often new C programmers will forget to use a compound statement for complex logic. The use of compound statements is seen in Figure 5-9. The first example shows a compound statement only for the true condition. The second example shows compound statements for both conditions. Note that the compound statements begin with an open brace and end with a close brace.



```
if (j != 3)
   {
    b++;
    printf("%d", b);
   } // if
else
    printf( "%d", j );
```

Compound statements are treated as one statement

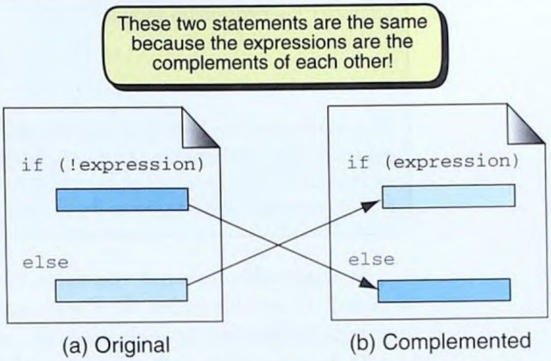
```
if (j != 5 && d == 2)
   {
    j++;
    d--;
    printf("%d%d", j, d);
   } // if
else
   {
    j--;
    d++;
    printf("%d%d", j, d);
   } // else
```

FIGURE 5-9 Compound Statements in an *if...else*

The sixth rule states that the true and false statements can be exchanged by complementing the expression. Recall from our discussion of relational operators on "Comparative Operators" in Section 5.1 that any expression can be complemented. When we find that we need to complement an *if...else* statement, all we have to do is to switch the true and false statements. An example of this operation is shown in Figure 5-10.

Null *else* Statement

Although there are always two possible actions after a decision, sometimes we do not care about both of them. In this case, the *else* action is usually left out. For example, assume we are processing numbers as they are being read. However, for some reason, the logic requires that we process only numbers greater than zero. As we read a number, we test it for greater than zero. If the test is true, we include it in the process. If it is false, we do nothing.



These two statements are the same because the expressions are the complements of each other!

```
if (!expression)

else
```

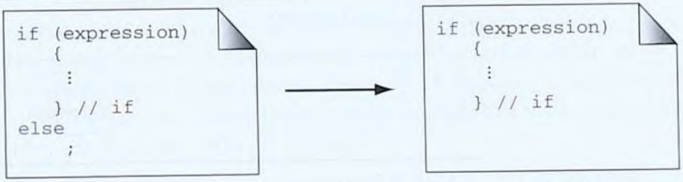
```
if (expression)

else
```

(a) Original

(b) Complemented

FIGURE 5-10 Complemented *if...else* Statements

If the *else* condition is not required—that is, if it is null—it can be omitted. This omission can be shown as a **null *else* statement** (a null statement consists of only a semicolon); more commonly, the *else* statement is simply omitted entirely, as shown in Figure 5-11.



```
if (expression)
    {
     :
    } // if
else
     ;
```

```
if (expression)
    {
     :
    } // if
```

FIGURE 5-11 A Null *else* Statement

De Morgan's Rule

When we design the logical flow of a program, we often have a situation in which the *not* operator is in front of a logical expression enclosed in the parentheses. Human engineering studies tell us, however, that positive logic is easier to read and understand than negative logic. In these cases, therefore, if we want to make the total expression positive by removing the parentheses, we apply the *not* operator directly to each operand. **De Morgan's rule** governs the complementing of operators in this situation. This rule is defined as follows:

When we remove the parentheses in a logical expression preceded by the *not* operator, we must apply the *not* operator to each expression while changing the logical operators—that is, changing *and* (&&) to *or* (||) and changing *or* (||) to *and* (&&).

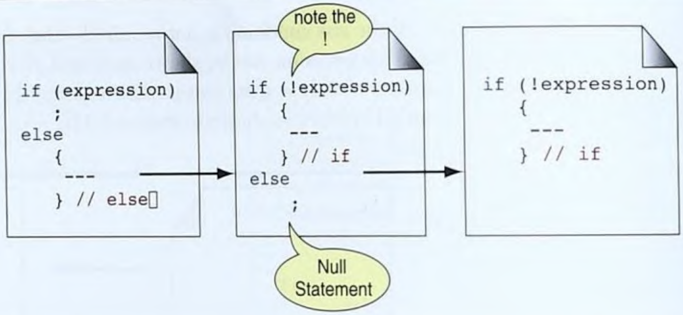
continued

Consider the expression shown below.

```
!(x && y) → !x || !y
!(x || y) → !x && !y
```

If the expression has been properly complemented according to De Morgan's rule, then the result of the first expression (*true* or *false*) will be the same as the result of the second expression for any given set of values for x and y. It is important to recognize, however, that in more complex situations, the C rules of precedence can affect De Morgan's rule.

It is possible to omit the false branch, but the true branch cannot be omitted. It can be coded as a null statement; normally, however, we do not use null in the true branch of an *if...else* statement. To eliminate it, we can use Rule 6 in Table 5-2, "Syntactical Rules for if...else Statements," which allows us to complement the expression and swap the two statements. The result is known as a **complemented *if...else*** and is shown in Figure 5-12.



```
if (expression)
   ;
else
   {
    ---
   } // else
```

```
if (!expression)
   {
    ---
   } // if
else
   ;
```

```
if (!expression)
   {
    ---
   } // if
```

FIGURE 5-12 A Null *if* Statement

Program 5-3 contains an example of a simple two-way selection. It displays the relationship between two numbers read from the keyboard.

PROGRAM 5-3 Two-way Selection

```
/* Two-way selection.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
   // Local Declarations
```

continued

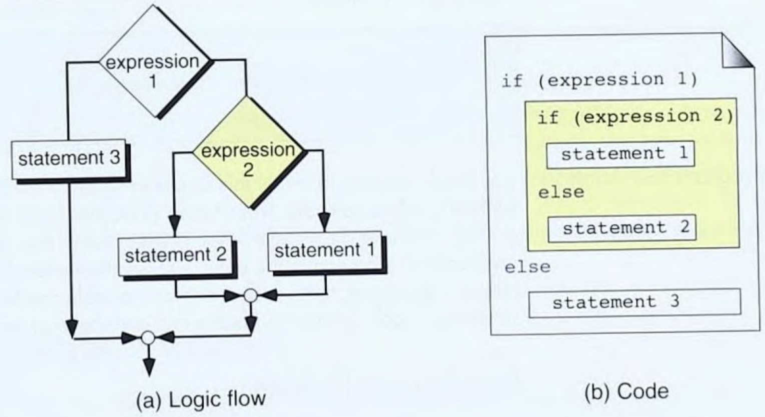
PROGRAM 5-3 Two-way Selection *(continued)*

```
   int a;
   int b;

   // Statements
   printf("Please enter two integers: ");
   scanf ("%d%d", &a, &b);

   if (a <= b)
       printf("%d <= %d\n", a, b);
   else
       printf("%d > %d\n", a, b);

   return 0;
} // main
```

```
Results:
Please enter two integers: 10 15
10 <= 15
```

Nested *if* Statements

As we stated previously, for the *if...else*, the statements may be any statement, including another *if...else*. When an *if...else* is included within an *if...else*, it is known as a **nested *if* statement**. Figure 5-13 shows a nested *if* statement. There is no limit to how many levels can be nested, but if there are more than three, they can become difficult to read.



```
if (expression 1)
   if (expression 2)
      statement 1
   else
      statement 2
else
   statement 3
```

FIGURE 5-13 Nested *if* Statements

Program 5-4 is a modification of Program 5-3. It uses a nested *if* statement to determine if a is less than, equal to, or greater than b.

PROGRAM 5-4 Nested *if* Statements

```
 1  /* Nested if in two-way selection.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6
 7  int main (void)
 8  {
 9  // Local Declarations
10     int a;
11     int b;
12
13  // Statements
14     printf("Please enter two integers: ");
15     scanf ("%d%d", &a, &b);
16
17     if (a <= b)
18        if (a < b)
19           printf("%d < %d\n", a, b);
20        else
21           printf("%d == %d\n", a, b);
22     else
23        printf("%d > %d\n", a, b);
24
25     return 0;
26  } // main
```

```
Results:
Please enter two integers: 10 10
10 == 10
```

Program 5-4 Analysis You should be able to follow this simple program with relative ease. However, it does contain a subtle software engineering principle. Study the *if* statements (17 and 18) carefully. Where do we check for *greater than*? The answer is that we default the *greater than* condition to the outer *else* (see Statement 22). When coding a two-way selection statement, try to code the most probable condition first; with nested selection statements, code the most probable first and the least probable last.

Dangling *else* Problem

Once we start nesting *if...else* statements, however, we encounter a classic problem known as the **dangling *else***. This problem is created when there is no matching *else* for every *if*. C's solution to this problem is a simple rule: *Always pair an* else *to the most recent unpaired* if *in the current block*. This rule may result in some *if* statement's being left unpaired. Since such an arbitrary rule often does not match our intent, we must take care to ensure that the resulting code is what we intended. Take, for instance, the example shown in Figure 5-14. From the code alignment, we conclude that the programmer intended the *else* statement to be paired with the first *if*. However, the compiler will pair it with the second *if*, as shown in the flowchart.

else is always paired with the most recent unpaired _if_.

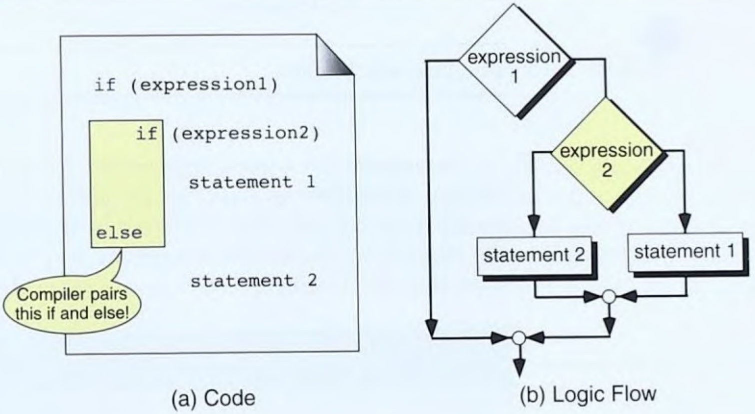
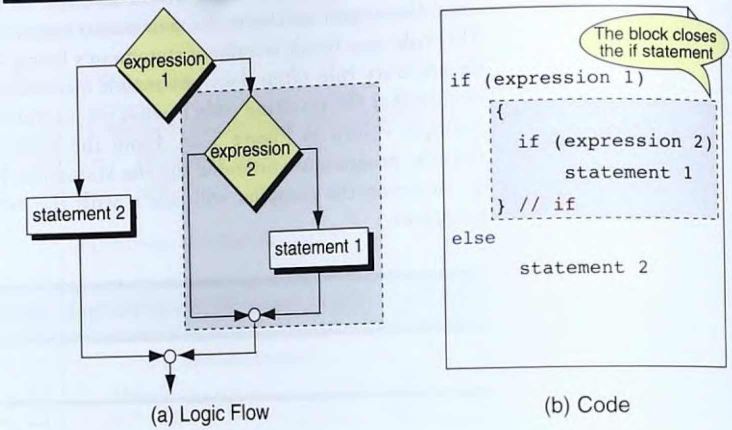


FIGURE 5-14 Dangling *else*

Figure 5-15 shows a solution to the dangling *else* problem, a compound statement. In the compound statement, we simply enclose the true actions in braces to make the second *if* a compound statement. Since the closing brace completes the body of the compound statement, the *if* statement is also closed from further consideration and the *else* is automatically paired with the correct *if*.

Simplifying *if* Statements

By now you should recognize that *if...else* statements can become quite complex. This discussion gives you some ideas on **simplifying if statements**. Usually, the purpose of simplification is to provide more readable code.



The block closes the if statement

```
if (expression 1)
  {
    if (expression 2)
        statement 1
  } // if
else
    statement 2
```

(a) Logic Flow

(b) Code

FIGURE 5-15 Dangling *else* Solution

Sometimes the control expression itself can be simplified. For example, the two statements in Table 5-3 are exactly the same. The simplified statements, however, are much preferred by experienced C programmers. When the simplified code becomes a natural way of thinking, you have begun to internalize the C concepts; that is, you are beginning to think in C!

Original Statement	Simplified Statement

`if (a != 0)`
`   statement`
	`if (a)`
	`   statement`

`if (a == 0)`
`   statement`
	`if (!a)`
	`   statement`

TABLE 5-3 Simplifying the Condition

Since the simplified statements in Table 5-3 are new, let's look at them a little more carefully. The expression `a != 0` evaluates to either *true* or *false*. If a is anything other than zero, then the expression is *true* and `statement1` is executed. However, any integer can be used to represent *true* or *false*. In this case, if a contains any value other than zero, it is *true*; otherwise, it is *false*. (See Figure 5-1.) Therefore, since we want to execute `statement2` whenever a is not zero, and since anything other than zero is *true*, we code the expression as `(a)`—that is, as "a is *true*." Similarly, if we want to test for a equal to zero, we simply complement the expression, making it `!a`.

Conditional Expressions

C provides a convenient alternative to the traditional *if...else* for two-way selection—the ternary conditional expression has a precedence of 3 (see inside front cover).

The **conditional expression** has three operands and a two-token operator. Each operand is an expression. The first token, a question mark (`?`), separates the first two expressions. The second token, a colon (`:`), separates the last two expressions. This gives it the following format:

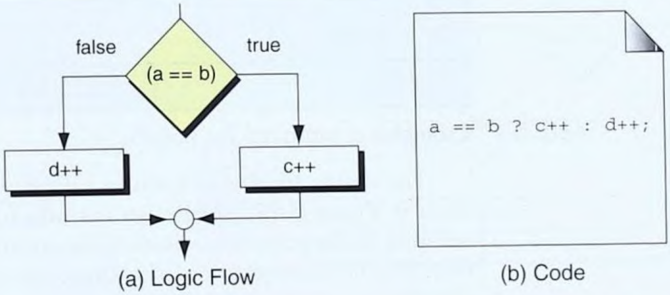
```
expression ? expression1 : expression2
```

To evaluate this expression, C first evaluates the leftmost expression. If the expression is *true*, then the value of the conditional expression is the value of `expression1`. If the expression is *false*, then the value of the conditional expression is the value of `expression2`.

Let's look at an example.

```
a == b ? c++ : d++;
```

In this expression, only one of the two side effects take place. If `a` is equal to `b`, `c++` is evaluated and one is added to `c`; `expression2` is ignored. On the other hand, if `a` is not equal to `b`, then `d++` is evaluated and 1 is added to `d`; `expression1` is ignored. If this sounds much like a simplified *if...else*, it's because it is! Figure 5-16 shows the flowchart for the expression, which could easily be coded as an *if...else*.



```
a == b ? c++ : d++;
```

(a) Logic Flow

(b) Code

FIGURE 5-16 Conditional Expression

The conditional expression, like any other expression, can be used in an assignment expression. Suppose that we have a program that can write either to a printer or to the system monitor. When we write to the monitor, we can write ten numbers to a line. When we write to the printer, we can write 15 numbers to a line. Given that `fileFlag` is a variable that indicates either the monitor (`M`) or printer (`P`), we could set the numbers per line as shown below. (The parentheses are not necessary but make the statement more readable.)

```
numPerLine = (fileFlag == 'M' ? 10 : 15);
```

One final note: Although we can nest conditional expressions, it is not recommended. When the logic begins to get complex, remember the KISS principle and use nested *if* statements.

Two-Way Selection Example

To demonstrate two-way selection, let's look at a program that calculates income taxes. A brief explanation of progressive tax brackets may be helpful. In a progressive bracket system, the higher the income, the higher the tax rate. However, the higher rates are applied only to the income in the bracket level. Thus, if you examine two incomes, they will both pay the same amount of taxes at the lower rates. This concept of marginal tax rates is shown in Table 5-4.

Case 1: Total Income 23,000			Case 2: Total Income 18,000		
Income in Bracket	**Tax Rate**	**Tax**	**Income in Bracket**	**Tax Rate**	**Tax**
(1) 10,000	2%	200	(1) 10,000	2%	200
(2) 10,000	5%	500	(2) 8,000	5%	400
(3) 3,000	7%	210	(3) none	7%	0
Total Tax		910	Total Tax		600

TABLE 5-4 Examples of Marginal Tax Rates

The design for the program to calculate taxes is shown in the structure chart in Figure 5-17. The design has only four functions besides *main*. The notation for `bracketTax` is somewhat unusual: In the final code, it is called five times in one expression. Therefore, we show it as a nested set of calls.

The design for each of the functions is shown in Figure 5-18. The logic for calculating the tax information requires three major steps: (1) Calculate the taxable income. Taxable income is total income adjusted for the number of dependents. In our example, we subtract $1,000 for each dependent.

(2) Calculate the taxes. The tax due calculation uses the progressive bracket system described at the beginning of the section. Our example uses five brackets.

(3) Calculate the taxes due or to be refunded. Note that if more taxes were withheld than are due, the taxes due is a negative number, indicating a refund.

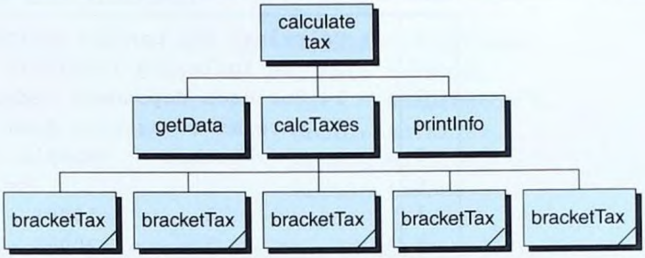


FIGURE 5-17 Design for Calculate Taxes

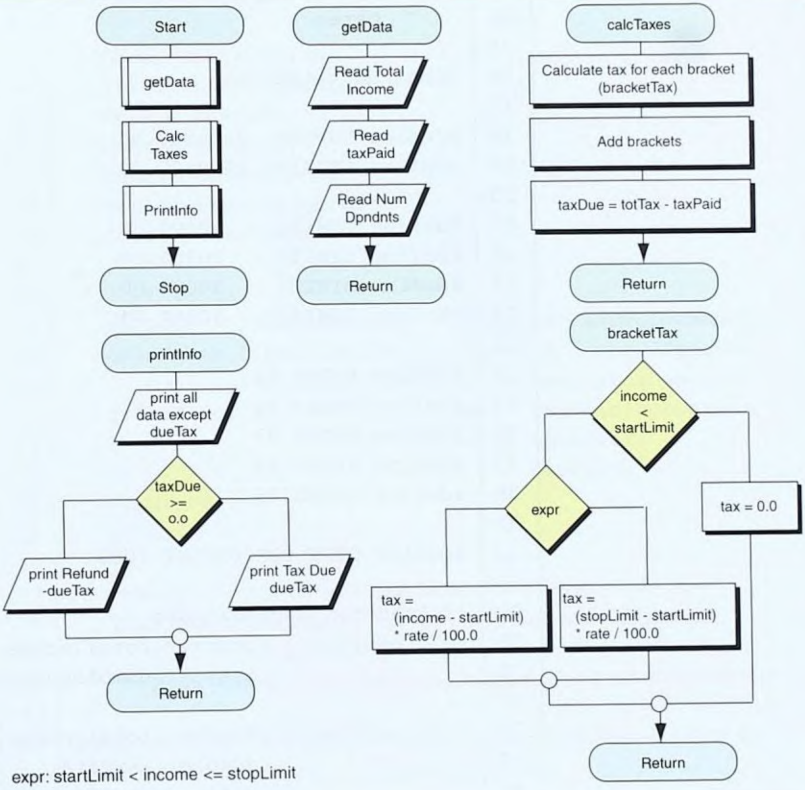


FIGURE 5-18 Design for Program 5-5

The code is seen in Program 5-5.

PROGRAM 5-5 Calculate Taxes

```
/* Calculate the tax due or the refund for a family based
   on the following imaginary formula.
   1. For each dependent deduct $1,000 from income.
   2. Determine tax rate from the following brackets:
           bracket    taxable income         tax rate
              1              <= 10000           2%
              2         10001 - 20000           5%
              3         20001 - 30000           7%
              4         30001 - 50000          10%
              5         50001 and up           15%
   Then print the amount of tax or the refund.

      Written by:
      Date:
*/
#include <stdio.h>

#define LOWEST  0000000.00
#define HIGHEST 1000000.00

#define LIMIT1    10000.00
#define LIMIT2    20000.00
#define LIMIT3    30000.00
#define LIMIT4    50000.00

#define RATE1 02
#define RATE2 05
#define RATE3 07
#define RATE4 10
#define RATE5 15

#define DEDN_PER_DPNDNT 1000

// Function Declarations
void getData   (double* totalIncome, double* taxPaid,
                int*    numOfDpndnts);

void calcTaxes (double  totalIncome,
                double  taxPaid,
                int     numOfDpndnts,
                double* taxableIncome,
                double* totalTax,
                double* taxDue);

```

continued

PROGRAM 5-5 Calculate Taxes *(continued)*

```
void printInformation (double totalIncome,
                       double taxPaid,
                       int    numOfDpndnts,
                       double totalTax,
                       double paidTax,
                       double taxDue);

double bracketTax (double income,
                   double startLimit,
                   double stopLimit,
                   int    rate);

int main (void)
{
// Local Declarations
   int    numOfDpndnts;
   double taxDue;
   double taxPaid;
   double totalIncome;
   double taxableIncome;
   double totalTax;

// Statements
   getData   (&totalIncome,  &taxPaid, &numOfDpndnts);
   calcTaxes (totalIncome,    taxPaid,  numOfDpndnts,
             &taxableIncome, &totalTax, &taxDue);
   printInformation (totalIncome,  taxableIncome,
                     numOfDpndnts, totalTax,
                     taxPaid,      taxDue);
   return 0;
}  // main

/* =================== getData ==================
   This function reads tax data from the keyboard.
      Pre  Nothing
      Post Reads totalIncome, taxPaid, & numOfDpndnts
*/
void getData ( double* totalIncome, double* taxPaid,
               int*    numOfDpndnts)
{
// Statements
   printf("Enter your total income for last year: ");
   scanf ("%lf", totalIncome);

```

continued

PROGRAM 5-5 Calculate Taxes *(continued)*

```
   printf("Enter total of payroll deductions      : ");
   scanf ("%lf", taxPaid);

   printf("Enter the number of dependents         : ");
   scanf ("%d", numOfDpndnts);
   return;
} // getData

/* ================ calcTaxes =================
   This function calculates the taxes due.
      Pre  Given-income, numOfDpndnts, & taxPaid
      Post Tax income, total tax, and tax due
           calculated
*/
void calcTaxes (double  totInc,
                double  taxPaid,
                int     numOfDpndnts,
                double* taxableInc,
                double* totTax,
                double* taxDue)
{
// Statements
   *taxableInc = totInc -
                       (numOfDpndnts* DEDN_PER_DPNDNT);
   *totTax =
        bracketTax(*taxableInc, LOWEST, LIMIT1,  RATE1)
      + bracketTax(*taxableInc, LIMIT1, LIMIT2,  RATE2)
      + bracketTax(*taxableInc, LIMIT2, LIMIT3,  RATE3)
      + bracketTax(*taxableInc, LIMIT3, LIMIT4,  RATE4)
      + bracketTax(*taxableInc, LIMIT4, HIGHEST, RATE5);

   *taxDue = *totTax - taxPaid;
   return;
} // calcTaxes

/* ================ printInformation =================
   This function prints a table showing all information.
      Pre  The parameter list
      Post Prints the table
*/
void printInformation (double totalIncome,
                       double income,
                       int    numDpndnts,
                       double totalTax,
```

continued

PROGRAM 5-5 Calculate Taxes *(continued)*

```
                         double paidTax,
                         double dueTax)
{
// Statements
   printf("\nTotal income           :%10.2f\n",
          totalIncome);
   printf("Number of dependents :%7d\n",   numDpndnts);
   printf("Taxable income         :%10.2f\n", income);
   printf("Total tax              :%10.2f\n", totalTax);
   printf("Tax already paid       :%10.2f\n", paidTax);

   if (dueTax >= 0.0)
       printf("Tax due               :%10.2f\n", dueTax);
   else
       printf("Refund                :%10.2f\n", -dueTax);
   return;
}  // printInformation

/* ================== bracketTax ==================
   Calculates the tax for a particular bracket.
      Pre    The taxableIncome
      Post   Returns the tax for a particular bracket
*/
double bracketTax (double income,    double startLimit,
                   double stopLimit, int    rate)
{
// Local Declarations
   double tax;

// Statements
   if (income <= startLimit)
       tax = 0.0;
   else
       if (income > startLimit && income <= stopLimit)
           tax = (income - startLimit) * rate / 100.00;
       else
         tax = (stopLimit - startLimit) * rate / 100.00;

   return tax;
}  // bracketTax
```

```
Results:
Enter your total income for last year: 15000
Enter total of payroll deductions: 250
```

continued

PROGRAM 5-5 Calculate Taxes *(continued)*

```
Enter the number of dependents: 2
Total income: 15000.00
Number of dependents: 2
Taxable income: 13000.00
Total Tax: 350.00
Tax already paid: 250.00
Tax due: 100.00
```

Program 5-5 Analysis Note that Program 5-5 contains extensive internal documentation. This documentation includes a series of comments at the beginning of the program, the define statements used to set some of the key values in the program, and the prototype statements.

Next examine the structure of the program; *main* contains no detail code; it simply calls three functions to get the job done. Since two of the functions must pass data back to *main*, they use the address (&) and indirection (*) operators. The call to `getData` in Statement 68 uses the address operator to pass the address of the three variables that need to be read from the keyboard. Then, Statements 82–83 in the `getData` header statement use the asterisk to specify that the type is an address. Because the parameters are already addresses, the *scanf* statements do not need an address operator. You must always consider what type of parameter you are using, data or address, and use the correct operators for it. Do not automatically use the address operator with *scanf.*

Examine the code for `calcTaxes`. The function header (starting at Statement 103) specifies that the first three formal parameters are passed as values and the last three are passed by address. The last three are address parameters because they are calculated values that need to be passed back to *main* for later printing. Note that all references to them in the function are prefaced with the indirection operator, which tells the compiler that it must use the variables in *main*.

Note the use of the type *double* throughout the program. If you examine the conversion specifications in the format strings, you will note that they are `lf` for input and `f` for output.

Finally, and the main point of this example, note how we used the function `bracketTax` to calculate the tax. This function was designed so that it could calculate the tax for any bracket. This is a much simpler design than writing complex code for different brackets and demonstrates how keeping it simple and short (KISS) makes for better programs.

5.3 Multiway Selection

In addition to two-way selection, most programming languages provide another selection concept known as **multiway selection**. Multiway selection chooses among several alternatives. The decision logic for the multiway statement is seen in Figure 5-19.

C has two different ways to implement multiway selection. The first is by using the *switch* statement. The other is a programming technique known as the *else-if* that provides a convenient style to nest *if* statements. The *switch* statement can be used only when the selection condition reduces to an integral expression. Many times, however, such as when the selection is based on a range of values, the condition is not an integral. In these cases, we use the *else-if*.

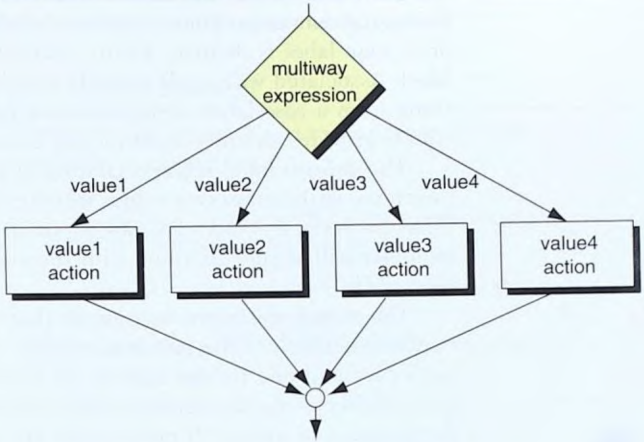


FIGURE 5-19 *switch* Decision Logic

The *switch* Statement

Switch is a composite statement used to make a decision between many alternatives. Figure 5-20 shows the *switch* statement syntax.[1]

```
switch (expression)
  {
   case constant-1: statement
                    :
                    statement
   case constant-2: statement
                    :
                    statement
   case constant-n: statement
                    :
                    statement
   default        : statement
                    :
                    statement
  } // end switch
```

FIGURE 5-20 *switch* Statement Syntax

Although the *switch* expression can use any expression that reduces to an integral value, the most common is a unary expression in the form of an

[1] Syntactically, the block in the *switch* statement is not needed if only one case is required. However, a simple *if* statement should be used if only one choice is available.

integral identifier. The selection alternatives, known as ***case* label**, must be C integral constants. For every possible value in the *switch* expression, a separate *case* label is defined. Figure 5-20 also shows the format for the *case* label. Associated with each possible *case* is zero or more statements. Everything from a *case* label to the next *case* label is a sequence. The *case* label simply provides an entry point to start executing the code.

The ***default* label** is a special form of the *case* label. It is executed whenever none of the other *case* values matches the value in the *switch* expression. Note, however, that *default* is not required. If we do not provide a default, the compiler will simply continue with the statement after the closing brace in the *switch*.

The *switch* statement is a puzzle that must be solved carefully to avoid confusion. Think of the *switch* statement as a series of drawbridges, one for each case and one for the default. As a result of the *switch* evaluation, one and only one of the drawbridges will be closed, so that there will be a path for the program to follow. (If none of the drawbridges is closed, then the statement is skipped and the program continues with the next statement after the *switch*.) The *switch* flow is shown in Figure 5-21.

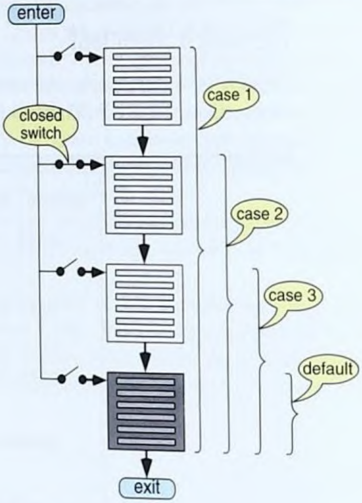


FIGURE 5-21 *switch* Flow

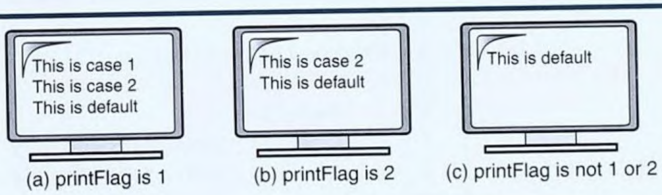
Pay careful attention to what happens once the program flow enters a *case* label. When the statements associated with one *case* have been executed, the program flow continues with the statements for the next *case*. In other words, once the program enters through a closed *switch*, it executes the code for all of the following *case*'s until the end. While we occasionally need this flexibility, it is not always what we want. We show how to break the flow shortly, but let's look at an example first. Program 5-6 demonstrates the *switch* statement. Can you figure out what it prints?

PROGRAM 5-6 Demonstrate the *switch* Statement

```
// Program fragment to demonstrate switch
   switch (printFlag)
      {
       case 1:  printf("This is case 1\n");

       case 2:  printf("This is case 2\n");

       default: printf("This is default\n");
      } // switch
```

Program 5-6 has three different *case* labels. The first *case* identifies the entry point to be used when `printFlag` is a 1. The second *case* identifies the entry point when `printFlag` is a 2. And finally, the *default* identifies the entry point when `printFlag` is neither a 1 nor a 2. While *default* is not a required condition in a *switch* statement, it should be included when all possible situations have not been covered by the *case* labels.

Have you figured out what is printed by Program 5-6? The answers are in Figure 5-22. Three results are possible, depending on the value in `printFlag`. If `printFlag` is a 1, then all three print statements are executed. If `printFlag` is a 2, then the first print statement is skipped and the last two are executed. Finally, if `printFlag` is neither a 1 nor a 2, then only the statements defined by the *default* are executed. This results in the first two print statements being skipped and only the last one being executed.



```
This is case 1
This is case 2
This is default
```

(a) printFlag is 1

```
This is case 2
This is default
```

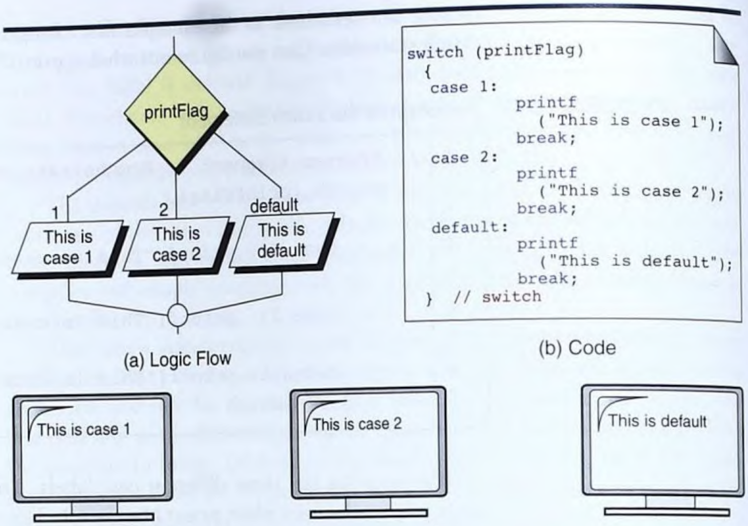
(b) printFlag is 2

```
This is default
```

(c) printFlag is not 1 or 2

FIGURE 5-22 *switch* Results

But what if we want to execute only one of the *case*-label sequences? To do so, we must use *break* statements. The ***break* statement** causes the program to jump out of the *switch* statement—that is, to go to the closing brace and continue with the code that follows the *switch*. If we add a *break* as the last statement in each *case*, we have Figure 5-23. Now, only one print statement will be executed, regardless of the value of `printFlag`.



(a) Logic Flow

```
switch (printFlag)
 {
  case 1:
             printf
               ("This is case 1");
             break;
  case 2:
             printf
               ("This is case 2");
             break;
  default:
             printf
               ("This is default");
             break;
 } // switch
```

(b) Code

FIGURE 5-23 A *switch* with *break* Statements

Two or more *case* labels can be associated with the same set of actions. In Program 5-7, for example, we print a message depending on whether `printFlag` is even or odd.

PROGRAM 5-7 Multivalued *case* Statements

```
/* Program fragment that demonstrates multiple
   cases for one set of statements
*/
switch (printFlag)
   {
    case 1:
    case 3:  printf("Good Day\n");
             printf("Odds have it!\n");
             break;
    case 2:
    case 4:  printf("Good Day\n");
             printf("Evens have it!\n");
             break;
    default: printf("Good Day, I'm confused!\n");
             printf("Bye!\n");
             break;
   } // switch
```

As a matter of style, the last statement in the *switch* does not require a *break*. We recommend, however, that you get in the habit of using it, especially when the last statement is not the *default*. This good habit will eventually save you hours of debugging time because you will not forget to add it when you add a new *case* to the switch statement.

Table 5-5 summarizes some points you must remember about the *switch* statement.

1. The control expression that follows the keyword *switch* must be an integral type.
2. Each *case* label is the keyword *case* followed by a constant expression.
3. No two *case* labels can have the same constant expression value.
4. But two *case* labels can be associated with the same set of actions.
5. The *default* label is not required. If the value of the expression does not match with any labeled constant expression, the control transfers outside of the *switch* statement. However, we recommend that all *switch* statements have a *default* label.
6. The *switch* statement can include at most one *default* label. The *default* label may be coded anywhere, but it is traditionally coded last.

TABLE 5-5 Summary of *switch* Statement Rules

switch Example

Program 5-8 converts a numeric score to a letter grade. The grading scale is the rather typical "absolute scale," in which 90% or more is an A, 80%–89% is a B, 70%–79% is a C, and 60%–69% is a D. Anything below 60% is an F.

PROGRAM 5-8 Student Grading

```
/* This program reads a test score, calculates the letter
   grade for the score, and prints the grade.
      Written by:
      Date:
*/
#include <stdio.h>

// Function Declarations
   char scoreToGrade (int score);

int main (void)
{
```

continued

PROGRAM 5-8 Student Grading *(continued)*

```
// Local Declarations
   int  score;
   char grade;

// Statements
   printf("Enter the test score (0-100): ");
   scanf ("%d", &score);

   grade = scoreToGrade (score);
   printf("The grade is: %c\n", grade);

   return 0;
}  // main
/* =================== scoreToGrade ===================
   This function calculates the letter grade for a score.
      Pre   the parameter score
      Post  returns the grade
*/
char scoreToGrade (int score)
{
// Local Declarations
   char grade;
   int  temp;

// Statements
   temp = score / 10;
   switch (temp)
       {
        case 10:
        case 9 : grade = 'A';
                 break;
        case 8 : grade = 'B';
                 break;
        case 7 : grade = 'C';
                 break;
        case 6 : grade = 'D';
                 break;
        default: grade = 'F';
       } // switch
   return grade;
}  // scoreToGrade
```

```
Results:
Enter the test score (0-100): 89
The grade is: B
```

Program 5-8 Analysis This example shows how we can use the integer division operator (/) to change a range of numbers to individual points to be used by the *switch* statement. The problem definition requires that if the score is between 80% and 89%, it must be changed to letter grade `'B'`. This condition cannot be used in a *switch* statement. But if we divide the score by 10 (integer division), the entire range (such as 80–89) can be changed to one single number (8), which can be used as a constant in the *case* label.

Note how the *break* statement works. This is an important part of the logic for *switch* statements. Without the *break*, we would have determined and assigned the score, and then proceeded to assign all of the lower scores down to `'F'`, with the result that everyone would have failed. The *break* allows us to leave the body of the *switch* as soon as we have completed the grade assignment.

One word of caution. If the user enters an invalid score, such as 110, this program gives invalid results. We describe how to prevent this problem in Chapter 6.

The *else-if*

The *switch* statement only works when the constant expression in the *case* labels are integral. What if we need to make a multiway decision on the basis of a value that is not integral? The answer is the *else-if*. There is no such C construct as the *else-if*. Rather, it is a style of coding that we use when we need a multiway selection based on a value that is not integral.

Suppose we need a selection based on a range of values. We code the first *if* condition and its associated statements, and then we follow it with all other possible values using *else-if*. The last test in the series concludes with an *else*. This is the default condition; that is, it is the condition that is to be executed if all other statements are false. A sample of the *else-if* logic design is shown in Figure 5-24.

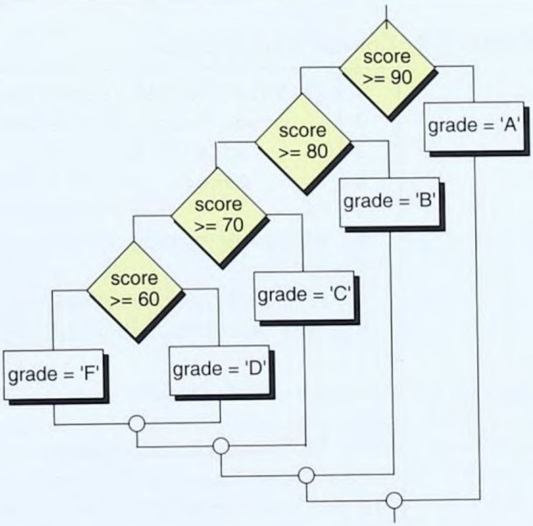


FIGURE 5-24 The *else-if* Logic Design for Program 5-9

What is different about the *else-if* code? As we said, it is really nothing more than a style change. Rather than indenting each *if* statement, we code the *else-if* on a single line and align it with the previous *if*. In this way we simulate the same formatting that we see in the *switch* and its associated *case* label. This style format is shown below.

```
if (score >= 90)
     grade = 'A';
else if (score >= 80)
     grade = 'B';
```

One important point about the *else-if*: Use it *only* when the same basic expression is being evaluated. In Figure 5-24, the expressions are all based on the variable score. If different variables were being evaluated, we would use the normal nesting associated with the *if...else* statement. Do not use the *else-if* format with nested *if* statements.

The *else-if* is an artificial C construct that is only used when

1. **The selection variable is not an integral, and**
2. **The same variable is being tested in the expressions.**

else-if Example

Program 5-9 is the same as the *switch* example in Program 5-8 but this time we use the *else-if* to solve the problem. It shows how we can use multiway selection and the *else-if* construct to change a numeric score to a letter grade.

PROGRAM 5-9 Convert Score to Grade

```
/* This program reads a test score, calculates the letter
   grade based on the absolute scale, and prints it.
      Written by:
      Date:
*/
#include <stdio.h>

// Function Declarations
char scoreToGrade (int score);

int main (void)
{
// Local Declarations
   int  score;
   char grade;

// Statements
```

continued

PROGRAM 5-9 Convert Score to Grade *(continued)*

```c
   printf("Enter the test score (0-100): ");
   scanf ("%d", &score);

   grade = scoreToGrade (score);
   printf("The grade is: %c\n", grade);

   return 0;
}  // main

/* ================== scoreToGrade ==================
   This function calculates letter grade for a score.
      Pre   the parameter score
      Post  returns the grade
*/
char scoreToGrade (int score)
{
// Local Declarations
   char grade;

// Statements
   if (score >= 90)
      grade = 'A';
   else if (score >= 80)
      grade = 'B';
   else if (score >= 70)
      grade = 'C';
   else if (score >= 60)
      grade = 'D';
   else
      grade = 'F';
   return grade;
}  // scoreToGrade
```

```
Results:
Enter the test score (0-100): 90
The grade is: A
```

Program 5-9 Analysis We used the *else-if* construct because our condition was not an integral; rather, it tested several ranges of the same variable, `score`. Study the code carefully. Note that, once the correct range is located, none of the following conditions will be tested. For instance, if a score of 85 is entered, the test against 90% is *false*, so we execute the *else-if* test for a score greater than 80%. Since this condition is *true*, we set grade to `'B'` and skip all the remaining tests.

Also, note how the tests are ordered. In this case we first eliminate those scores equal to or greater than 90%; then we check 80%, 70%, and 60% in turn. Since we were checking for greater than, we could not have coded it in the reverse, 60% first.

This is an important design concept: When checking a range using greater than, start with the largest value; when checking a range using less than, start with the lowest value.

5.4 More Standard Functions

One of the assets of the C language is its rich set of standard functions that make programming much easier. In Chapter 4 we introduced some of these standard functions. Now that we have studied selection, we can discuss two other libraries of standard functions that are closely related to selection statements.

C99 has two parallel but separate header files for manipulating characters. The first, *ctype.h*, supports the ASCII character set. The second, *wctype.h*, supports wide characters. While the wide-character library (*wctype.h*) contains some functions not found in the ASCII character library (*ctype.h*), they follow a common naming format. For example, the ASCII library contains a function, *islower*, to test for lowercase alphabetic characters. The equivalent function in the wide-character library is *iswlower*. Therefore, if we know the name for one library, we know it for the other.

Standard Characters Functions

The character libraries are divided into two major groups: classifying functions and converting functions. The prototypes of these functions are in the *ctype.h* or *wctype.h* header files. Before looking at these functions, make sure you understand the classification of characters that C uses. This breakdown of classes is shown in Figure 5-25, which uses a tree to show how characters are classified. You read the tree much like a structure chart, starting at the top and following the branches to the bottom.

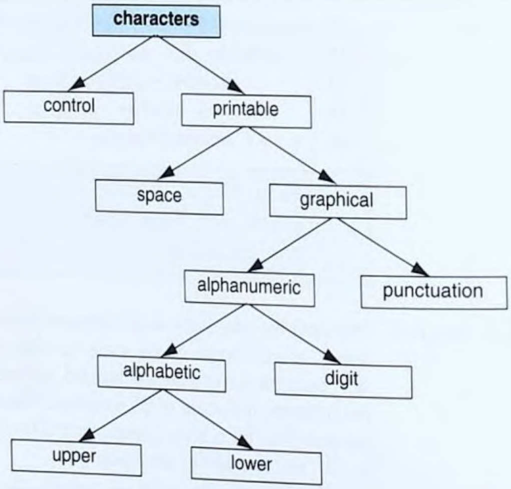


FIGURE 5-25 Classifications of the Character Type

Characters are first broken down into control characters, such as carriage return and end of file, or into printable characters. This tells us that control characters are not printable. The printable characters are either a space or the rest of the printable characters, which are classified as graphical. In turn, the graphical characters are broken down into alphanumeric and punctuation characters. Alphanumeric means either an alphabetic character or a digit. Finally, alphabetic characters are either upper- or lowercase.

Classifying Functions

Classifying functions examine a character and tell if it belongs to a given classification as described previously. They all start with the prefix *is* and return *true* if the actual parameter is in the specified class and *false* if it is not. The prototypes of these functions are found in the *ctype.h* and *cwtype.h* files.

The general form of the prototype function is

```
int  is… (int testChar)
```

where the function name starts with `is` or `isw`;[2] for example, `iscntrl`, which stands for "is a control character."

All of the classifying functions return *true* or *false*. If the character matches the set being tested by the function, it returns *true*.; if it doesn't, it returns *false*. For example, the *isdigit* function tests the character against the decimal digits (0 through 9). If the character is a decimal digit, it returns *true*; if the character is not a decimal digit, it returns *false*. Table 5-6 summarizes each function with a brief explanation.

Function	Description
iscntrl	Control characters
isprint	Printable character, that is character with an assigned graphic
isspace	Whitespace character: space character (32), horizontal tab (9), line feed (10), vertical tab (11), form feed (12), and carriage return (13)
isgraph	Character with printable graphic; all printable characters except space
isalnum	Alphanumeric: any alphabetic or numeric character
ispunct	Any graphic character that is not alphanumeric
isalpha	Any alphabetic character, upper- or lowercase

continued

TABLE 5-6 Classifying Functions

[2] For wide-character functions.

Function	Description
isupper	Only uppercase alphabetic
islower	Only lowercase alphabetic
isdigit	Decimal digits (0...9)
isxdigit	Hexadecimal digits (0...9, a...f, A...F)
isodigit	Octal digits (0...7)

TABLE 5-6 Classifying Functions *(continued)*

Character Conversion Functions

Two **converting functions** are used to convert a character from one case to another. These functions start with prefix `to` or `tow` (for wide characters) and return an integer that is the value of the converted character. Their basic format is

```
int to… (int oldChar)
```

Table 5-7 summarizes each function with a brief explanation.

Function	Description
toupper	Converts lower- to uppercase. If not lowercase, returns it unchanged.
tolower	Converts upper- to lowercase. If not uppercase, returns it unchanged.

TABLE 5-7 Conversion Functions

A Classification Program

Let's write a program that uses classification functions to examine a character input from the keyboard. If you study Figure 5-25, carefully, you will note that most characters fall into more than one classification. For instance, a digit is printable, graphical, alphanumeric, and a digit.

Our program tests from the bottom up, so that only one classification will be printed for each character. For a digit, only the fact that it is a digit will be printed. Because it is a relatively simple demonstration of the classification functions, we wrote it using only *main*. The solution is shown in Program 5-10.

PROGRAM 5-10 Demonstrate Classification Functions

```c
/* This program demonstrates the use of the character
   classification functions found in the c-type library.
   Given a character, it displays the highest
   classification for the character.
      Written by:
      Date:
*/

#include <stdio.h>
#include <ctype.h>

int main (void)
{
// Local Declarations
   char charIn;

// Statements
   printf("Enter a character to be examined: ");
   scanf ("%c", &charIn);

   if (islower(charIn))
       printf("You entered a lowercase letter.\n");
   else if (isupper(charIn))
       printf("You entered an uppercase character.\n");
   else if (isdigit(charIn))
       printf("You entered a digit.\n");
   else if (ispunct(charIn))
       printf("You entered a punctuation character.\n");
   else if (isspace(charIn))
       printf("You entered a whitespace character.\n");
   else
       printf("You entered a control character\n");
   return 0;
}  // main
```

```
Results:
Enter a character to be examined: a
You entered a lowercase letter.
```

Program 5-10 Analysis Note the use of the *else-if* in this program. While it may not appear so at first analysis, this is actually a range analysis problem, which makes it suitable for multiway selection.

Handling Major Errors

One of the better known computer acronyms is GIGO—garbage in, garbage out. In writing programs, we must decide how to handle errors to prevent garbage from corrupting the data. Sometimes, as we will see in Chapter 6, we can recover from the error by having the user re-enter it. Other times, there is no way to recover.

When we can't recover, C provides two functions that allow us to terminate the functions: *exit* and *abort*. Both functions are found in the standard library (*stdlib.h*).

exit

Whereas *return* terminates a function, *exit* terminates the program regardless of where in the program it is executed. While we use it to terminate the program because we detected an error, C considers it a normal termination. For this reason, the termination is orderly; any pending file stream writes are first completed and all streams are closed. The exit prototype statement is shown below.

```
void exit (int terminationStatus);
```

There is one parameter to the *exit* call, an integer value to be passed to the operating system. While any integer is acceptable, it is usually a nonzero, indicating that the program did not complete successfully. We demonstrate the exit function in Program 5-12, "Menu-driven Calculator—Third Increment."

abort

The *abort* function is used to terminate a program abnormally. It is considered a non-orderly termination because the output streams are not flushed and they are not closed. It is like drawing the "Go to Jail" card in Monopoly—you go directly to jail, you do not pass GO, and you do not collect $200. When *abort* is called, the program immediately goes to the operating system.

The abort function has no parameters. Its prototype statement is shown below.

```
void abort (void);
```

5.5 Incremental Development Part II

In Chapter 4, we introduced the concept of incremental development with a simple calculator program. We continue the discussion by adding a menu and calculator subfunctions.

Calculator Design

Although it is an elementary example, our calculator program illustrates two concepts. First, it shows how we can communicate with a user through a menu. Menu's are a common design for programs that interact with users. Second, it demonstrates incremental development through three levels in a program. Figure 5-26 shows the design of the program.

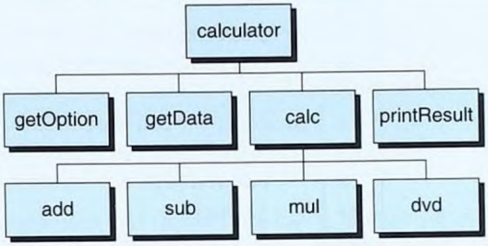


FIGURE 5-26 Design for Menu-driven Calculator

Calculator Incremental Design

Good structured programming requires that each function do only one thing. In this program, we are doing four different things: (1) We must ask the user what function is desired. Then we need to (2) get the data for the operation, (3) make the calculation, and finally (4) print the result. These four processes are seen as called functions in *main*. In its turn, `calc` calls one of four functions to perform an arithmetic operation. Finally, the print result function displays the result.

First Increment: *main* and Get Option

Program 5-11 begins the development with the *main* function and the get option function.

PROGRAM 5-11 Menu-driven Calculator—First Increment

```
/* This program uses a menu to allow the user to add,
   multiply, subtract, or divide two integers.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>

// Function Declarations
```

continued

PROGRAM 5-11 Menu-driven Calculator—First Increment *(continued)*

```
int    getOption    (void);

int main (void)
{
// Local Declarations
   int   option;

// Statements
   option = getOption();

printf("**You selected option %d\n", option);

   return 0;
}  // main

/* ==================== getOption ====================
   This function shows a menu and reads the user option.
      Pre    Nothing
      Post   returns the option
*/
int getOption (void)
{
// Local Declarations
   int option;

// Statements
   printf  ("\t*********************************");
   printf("\n\t*            MENU               *");
   printf("\n\t*                               *");
   printf("\n\t*  1. ADD                       *");
   printf("\n\t*  2. SUBTRACT                  *");
   printf("\n\t*  3. MULTIPLY                  *");
   printf("\n\t*  4. DIVIDE                    *");
   printf("\n\t*                               *");
   printf("\n\t*********************************");

   printf("\n\nPlease type your choice ");
   printf("and key return: ");
   scanf ("%d", &option);
printf("**You selected option %d\n", option);
   return option;
}  // getOption
```

```
Results:
   *********************************
```

continued

PROGRAM 5-11 Menu-driven Calculator—First Increment *(continued)*

```
     *              MENU              *
     *                                *
     *   1. ADD                       *
     *   2. SUBTRACT                  *
     *   3. MULTIPLY                  *
     *   4. DIVIDE                    *
     *                                *
     **********************************

 Please type your choice and key return: 3
 **You selected option 3
 **You selected option 3
```

Program 5-11 Analysis We use the same approach that we used previously. In *main* we call the get option function. Just before returning, we print the option and then we re-print it in *main* so that we can easily verify the results.

We incorporate one new style. Because it can be difficult to find the debugging statements in a large function, we do not indent them in the function. Rather we leave them flush to the left margin so that they are easy to find and remove and we continue the incremental development.

Second Increment: getData

Following the structure chart, we next write and debug get data. Because we wrote and debugged the get data function in Chapter 4, we do not need to include it here. We simply copy it into our new program.

Third Increment: Calculate

The third function is calculate (`calc`). This is the first example of a subfunction that calls other subfunctions: add, subtract, multiply, and divide. In this case, we write the complete function except for the subfunction calls.

The design for the calculate function requires a *switch* statement to select the correct subfunction. In place of the actual calls, we return dummy values: 1.0 for add, 2.0 for subtract, 3.0 for multiply, and 4.0 for divide. You should recognize these as the option numbers. When testing a program, use test data that is easy to remember and that can be easily verified. The code is shown in Program 5-12.

PROGRAM 5-12 Menu-driven Calculator—Third Increment

```
/* This program uses a menu to allow the user to add,
   multiply, subtract, or divide two integers.
      Written by:
      Date:
*/
```

continued

PROGRAM 5-12 Menu-driven Calculator—Third Increment *(continued)*

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

// Function Declarations
int   getOption (void);
void  getData   (int* num1, int* num2);
float calc      (int option, int num1, int num2);

int main (void)
{
// Local Declarations
   int   option;
   int   num1;
   int   num2;
   float result;

// Statements
   option = getOption();
   getData (&num1, &num2);
   result = calc (option, num1, num2);
printf("**In main result is: %6.2f", result);

   return 0;
}  // main

/* ==================== getOption =====================
   This function shows a menu and reads the user option.
      Pre    Nothing
      Post   returns the option
*/
int getOption (void)
{
// Local Declarations
   int option;

// Statements
   printf  ("\t************************************");
   printf("\n\t*             MENU                 *");
   printf("\n\t*                                  *");
   printf("\n\t*  1. ADD                          *");
   printf("\n\t*  2. SUBTRACT                     *");
   printf("\n\t*  3. MULTIPLY                     *");
   printf("\n\t*  4. DIVIDE                       *");
   printf("\n\t*                                  *");
```

continued

PROGRAM 5-12 Menu-driven Calculator—Third Increment *(continued)*

```
   printf("\n\t**********************************");

   printf("\n\nPlease type your choice ");
   printf("and key return: ");
   scanf ("%d", &option);
   return option;
}  // getOption

/* ===================== getData ====================
   This function reads two integers from the keyboard.
      Pre   Parameters a and b are addresses
      Post  Data read into parameter addresses
*/
void getData (int* a, int* b)
{
   printf("Please enter two integer numbers: ");
   scanf("%d %d", a, b);
   return;
}  // getData

/* =================== calc ====================
   This function determines the type of operation
   and calls a function to perform it.
      Pre   option contains the operation
            num1 & num2 contains data
      Post  returns the results
*/
float calc (int option, int num1, int num2)
{
// Local Declarations
float result;

// Statements
printf("**In calc input is: %d %d %d\n",
option, num1, num2);
   switch(option)
       {
        case 1:  result = 1.0;                // Add
                 break;
        case 2:  result = 2.0;                // Subtract
                 break;
        case 3:  result = 3.0;                // Multiply
                 break;
        case 4:  if (num2 == 0.0)             // Divide
```

continued

PROGRAM 5-12 Menu-driven Calculator—Third Increment *(continued)*

```
                   {
                    printf("\n\a\aError: ");
                    printf("division by zero ***\n");
                    exit (100);
                   } // if
                 else
                    result = 4.0;
                 break;
        /* Better structured programming would validate
           option in getOption. However, we have not
           yet learned the technique to code it there.
        */
        default: printf("\aOption not available\n");
                 exit (101);
      } // switch
printf("**In calc result is: %6.2f\n", result);
   return result;
   return result;
} // calc
```

```
Results:
    *********************************
    *           MENU                *
    *                               *
    *  1. ADD                       *
    *  2. SUBTRACT                  *
    *  3. MULTIPLY                  *
    *  4. DIVIDE                    *
    *                               *
    *********************************

Please type your choice and key return: 1
Please enter two integer numbers: 13 8
**In calc input is: 1 13 8
**In calc result is:   1.00
**In main result is:   1.00
```

Program 5-12 Analysis Once again, note how we verify both the downward and upward communication. The calculate function displays the parameter values it receives and the result before it returns. In *main* the value that was returned is displayed.

Before we can begin to write the subfunctions, however, we must run four more tests, one for subtract, one for multiply, and two for divide. To test divide, we must first test it with a zero divisor and then with a valid divisor.

Fourth Increment: Add

We wrote the code for the add function in Chapter 4 so we don't need to write and debug it again. We can simply copy it to our new program. (Because of the design change required to pass back a real number from divide, we must change the add function to return a *float* rather than an *int*. This change is minor, however, so we don't show the incremental program for it here.

Fifth Increment: Subtract

At this point we have verified that the calculate parameters are passed correctly and the correct value is being returned. We are therefore ready to develop and debug its subfunctions. Note that the debugging code is still in place for the multiply and divide functions. We do not remove it until the subfunctions have been written and debugged.

On the other hand, we have removed the input parameter displays because they have been debugged. The upward communication verification display remains, however, until the calculate function is complete. The code is shown in Program 5-13.

PROGRAM 5-13 Menu-driven Calculator—Fifth Increment

```
/* This program uses a menu to allow the user to add,
   multiply, subtract, or divide two integers.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>

// Function Declarations
int   getOption (void);
void  getData   (int* num1,  int* num2);
float calc      (int option, int num1, int num2);
float add       (int num1,   int num2);
float sub       (int num1,   int num2);

int main (void)
{
// Local Declarations
   int   option;
   int   num1;
   int   num2;
   float result;

```

continued

PROGRAM 5-13 Menu-driven Calculator—Fifth Increment *(continued)*

```
// Statements
   option = getOption();
   getData (&num1, &num2);
   result = calc (option, num1, num2);
// Temporary code--to be removed
printf("**In main result is: %6.2f\n", result);

   return 0;
}  // main

/* ==================== getOption ====================
   This function shows a menu and reads the user option.
      Pre   Nothing
      Post  returns the option
*/
int getOption (void)
{
// Local Declarations
   int option;

// Statements
   printf  ("\t************************************");
   printf("\n\t*              MENU                *");
   printf("\n\t*                                  *");
   printf("\n\t*  1. ADD                          *");
   printf("\n\t*  2. SUBTRACT                     *");
   printf("\n\t*  3. MULTIPLY                     *");
   printf("\n\t*  4. DIVIDE                       *");
   printf("\n\t*                                  *");
   printf("\n\t************************************");

   printf("\n\nPlease type your choice ");
   printf("and key return: ");
   scanf ("%d", &option);
   return option;
}  // getOption

/* ===================== getData =====================
   This function reads two integers from the keyboard.
      Pre   Parameters a and b are addresses
      Post  Data read into parameter addresses
*/
void getData (int* a, int* b)
{
   printf("Please enter two integer numbers: ");
```

continued

PROGRAM 5-13 Menu-driven Calculator—Fifth Increment *(continued)*

```
   scanf("%d %d", a, b);
   return;
}  // getData

/* ==================== calc ====================
   This function determines the type of operation
   and calls a function to perform it.
      Pre   option contains the operation
            num1 & num2 contains data
      Post  returns the results
*/
float calc (int option, int num1, int num2)
{
// Local Declarations
float result;

// Statements
   switch(option)
       {
        case 1:  result = add (num1, num2);
                 break;
        case 2:  result = sub (num1, num2);
                 break;
        case 3:  result = 3.0;               // Multiply
                 break;
        case 4:  if (num2 == 0.0)            // Divide
                     {
                      printf("\n\a\aError: ");
                      printf("division by zero ***\n");
                      exit (100);
                     } // if
                 else
                     result = 4.0;
                 break;
        /* Better structured programming would validate
           option in getOption. However, we have not
           yet learned the technique to code it there.
        */
        default: printf("\aOption not available\n");
                 exit (101);
       } // switch
printf("**In calc result is: %6.2f\n", result);
   return result;
}  // calc
```

continued

PROGRAM 5-13 Menu-driven Calculator—Fifth Increment *(continued)*

```c

/* ===================== add ======================
   This function adds two numbers and returns the sum.
      Pre   a and b contain values to be added
      Post  Returns a + b
*/
float add (int a, int b)
{
// Local Definitions
   float sum;

// Statements
   sum = a + b;
   return sum;
}  // add

/* ===================== sub ======================
   This function subtracts two numbers
      Pre   a and b contain values to be subtracted
      Post  Returns a + b
*/
float sub (int a, int b)
{
// Local Definitions
   float dif;

// Statements
   dif = a - b;
printf("**In sub result is:  %6.2f\n", dif);
   return dif;
}  // sub
```

```
Results:
   *********************************
   *              MENU             *
   *                               *
   *  1. ADD                       *
   *  2. SUBTRACT                  *
   *  3. MULTIPLY                  *
   *  4. DIVIDE                    *
   *                               *
   *********************************

Please type your choice and key return: 2
```

continued

PROGRAM 5-13 Menu-driven Calculator—Fifth Increment *(continued)*

```
Please enter two integer numbers: 13 8
**In sub result is:    5.00
**In calc result is:   5.00
**In main result is:   5.00
```

Program 5-13 Analysis Note how we display the results up the program to *main*. This verifies that the upward communication is correct. If there should be a problem, we will know exactly where it occurred.

Remaining Increments

At this point, we have demonstrated the incremental development concepts for programs using multiple levels of subprograms. To complete the program, we must write and debug the final three increments:

- Increment 6: multiply
- Increment 7: divide
- Increment 8: print results

We leave the completion of the program for you as a problem at the end of the chapter.

5.6 Software Engineering

Dependent Statements

Several statements in the C language control other statements that follow them. The *if...else* is the first of these statements that we have seen. Whenever one statement controls or influences statements that follow it, good structured programming style indents the **dependent statements** to show that the indented code is dependent on the controlling statement. The compiler does not need the indentation—it follows its syntactical rules regardless of how a program is formatted—but good style makes for readable programs.

To illustrate the point, consider the two versions of the code for the function in Program 5-14. They both accomplish the same task. To make this exercise even more meaningful, cover up the right half of the program and predict the results that will be produced when the ill-formed code executes. Then look at the well-structured code.

PROGRAM 5-14 Examples of Poor and Good Nesting Styles

```
             Poor Style                         Good Style
 1  int someFun (int a, int b)    int someFun (int a, int b)
 2  {                             {
 3  int x;                           int x;
 4
 5     if (a < b)                    if (a < b)
 6     x = a;                           x = a;
 7     else                          else
 8        x = b;                        x = b;
 9        x *= .5f;                  x *= .5f;
10     return x;                     return x;
11  }  // someFun                 }  // someFun
```

Assume that in this example, a has a value of 10 and b has a value of 20. What value will be returned? First, look at Statement 6. The assignment of x in this example is dependent on the *if* in Statement 5. Since Statement 6 is not indented, however, it is difficult to see the dependency. Because the value of a (10) is less than the value of b (20), x will be assigned the value 10.

Now examine Statement 9. It is indented and therefore appears to be dependent on the *else* statement. But is it? The answer is no. It just looks that way and is therefore misleading. Statement 9 will therefore execute regardless of the expression in the *if* statement. This relationship is much more clearly seen in the well-styled code on the right. The code on the right is properly indented to show the relationships among the statements, and therefore the chance of misreading the code is minimal.

The indentation rules are summarized in Table 5-8.

1. Indent statements that are dependent on previous statements. The indentations are at least three spaces from the left end of the controlling statement.
2. Align *else* statements with their corresponding *if* statements. (See Figure 5-13.)
3. Place the opening brace identifying a body of code on a separate line. Indent the statements in the body of the code one space to the right of the opening brace.
4. Align the closing brace identifying a body of code with the opening brace, and place the closing brace on a separate line. Use a comment to identify the block being terminated.
5. Align all code on the same level, which is dependent on the same control statement.
6. Further indent nested statements according to the above rules.
7. Surround operators with whitespace.
8. Code only one definition or statement on a single line.
9. Make your comments meaningful at the block level. Comments should not simply parrot the code.

TABLE 5-8 Indentation Rules

Negative Logic

Negative logic refers to any expression that begins with *not* or that contains multiple *not* expressions within. Like the proverbial double-negative in the English language, interpreting negative logic can be difficult. We therefore try to avoid it.

In the discussion of Figure 5-12 and in the section on simplifying the *if* statement, one technique that we proposed was complementing a conditional statement. This requires making a positive statement negative and a negative statement positive. This can be done by following Rule 6 in Table 5-2, "Syntactical Rules for if…else Statements," which states that the positions of the statements in an *if…else* can be swapped if the original control expression is complemented. The concept of complementing the *if* statement was shown in Figure 5-10, "Complemented *if...else* Statements."

Remember, however, that simple code is the clearest code. This concept has been formulated into an acronym: KISS, which stands for "Keep It Simple and Short!" (see Chapter 3). Unfortunately, negative logic is not always simple. In fact, it can get extremely confusing. We have seen professional programmers work for hours trying to debug negative logic.

Avoid compound negative statements!

When you complement an expression, make sure that the resulting statement is readable. Complementing an expression can be more difficult than simply making the condition negative. Examine the third statement in Table 5-9 carefully. Note that the complement of the *not* (!) is *not-not* (! !), which in effect cancels the *not*. In general, you should avoid compound negative statements. In this case, therefore, the complemented statement is greatly preferred.

Original Statement	Complemented Statement
`if (x <= 0)`	`if (x > 0)`
`if (x != 5)`	`if (x == 5)`
`if (!(x <= 0 || !flag))`	`if (x > 0 && flag)`

TABLE 5-9 Complementing Expressions

Rules for Selection Statements

For selection statements, you need to consider three other rules, shown in Table 5-10. Since these rules are sometimes conflicting, they are listed in their order of importance.

1. Code positive statements whenever possible.
2. Code the normal/expected condition first.
3. Code the most probable conditions first.

TABLE 5-10 Selection Rules

Human engineering studies have shown that people make fewer errors when reading positive statements than when reading negative statements. This is especially true when complex, compound Boolean statements are involved. Therefore, the first rule in Table 5-10 is, whenever possible, code your selection statements using positive conditions.

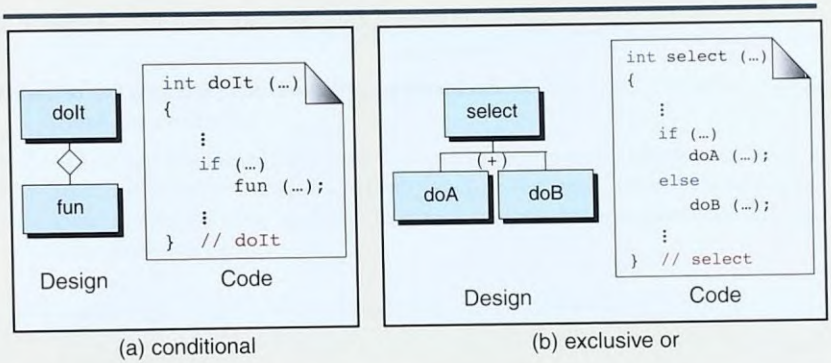
The second rule concerns the human expectations about what will follow. People have a tendency to anticipate things. They will therefore be less confused if what follows is what they expect. In most cases, this means coding the anticipated condition first.

Finally, the third rule concerns the efficiency of the resulting program. Coding the most probable conditions first is especially important in a multiway selection, such as the *else-if*. When you code the most probable test first, then the program can skip the rest of the statements. Obviously, the more statements skipped, the more efficient the resulting program.

As we mentioned previously, these rules often conflict with each other. We have listed them in their order of importance from a human engineering point of view. Unless there are overriding circumstances, you should select the higher option (Rule 1 before Rule 2 before Rule 3) in case of conflicts. But remember the overriding principle: KISS—Keep It Simple and Short.

Selection in Structure Charts

We introduced the basic concepts of structure charts in Chapter 4. In this section we extend the discussion to **selection in structure charts**. Figure 5-27 shows two symbols for a function that is called by a selection statement, the condition and the exclusive *or*.



```
int doIt (…)
{
    ⋮
    if (…)
       fun (…);
    ⋮
}  // doIt
```

(a) conditional

```
int select (…)
{
   ⋮
   if (…)
      doA (…);
   else
      doB (…);
   ⋮
}  // select
```

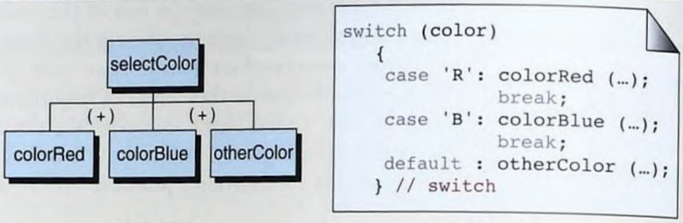
(b) exclusive or

FIGURE 5-27 Structure Chart Symbols for Selection

In Figure 5-27(a), the function `doIt` contains a conditional call to a subfunction, `fun`. If the condition in the *if* statement is *true*, we call `doIt`. If it is not *true*, we skip `doIt`. This situation is represented in a structure chart as a small diamond on the vertical line between the two function blocks.

Figure 5-27(b) represents the selection between two different functions. In this example, the function `select` chooses between `doA` and `doB`. One and only one of them will be called each time the conditional statement is executed. This is known as an **exclusive *or***; one of the two alternatives is executed to the exclusion of the other. The exclusive or is represented by a plus sign between the processes.

Now consider the design found when a series of functions can be called exclusively. This occurs when a multiway selection contains calls to several different functions. Figure 5-28 contains an example of a *switch* statement that calls different functions based on color.



```
switch (color)
   {
    case 'R': colorRed (…);
              break;
    case 'B': colorBlue (…);
              break;
    default : otherColor (…);
   } // switch
```

(a) Design

(b) Code

FIGURE 5-28 Multiway Selection in a Structure Chart

The structure chart rules described in this section follow:

1. Conditional calls are indicated by a diamond above the called function rectangle.
2. Exclusive or calls are indicated by a (+) between functions.

5.7 Tips and Common Programming Errors

1. The complement of < is >=, and the complement of > is <=.
2. Dangling *else* statements are easily created and difficult to debug. One technique for avoiding dangling *else* statements is to use braces, even when they are not needed.
3. The expression in the control expression in the *if...else* statement may have a side effect, as shown below.

```
if (a++)
```

4. Encapsulate the statements inside braces if you have more than one statement after *if* or *else* in an *if...else* statement.
5. Do not use the equal operator with a floating-point number. Floating-point numbers are seldom exactly equal to a required value. When you need to test for equality, such as a == b, use the expression shown below.

```
if (fabs (a - b) < .0000001)
```

6. Do not forget to use a *break* statement when the cases in a *switch* statement are exclusive.
7. While not necessarily an error, it is poor programming practice to write a *switch* statement without a *default* label. If the logic doesn't require one, code it with an error message to guard against unanticipated conditions. This is shown below.

```
default: printf("\aImpossible default\n");
         exit (100);
```

8. The most common C error is using the assignment operator (=) in place of the equal operator (==). One way to minimize this error is to get in the habit of using the term "assignment operator" when reading the code. For example, say "a is assigned b," not "a equals b."
9. It may be an error to place a semicolon after the *if* expression.
   a. The semicolon terminates the *if* statement, and any statement that follows it is not part of the *if*.
   b. It is a compile error to code an *else* without a matching *if*. This error is most likely created by a misplaced semicolon.

```
if (a == b);                        // if terminated here
   printf(…);
else                               // No matching if
   printf(…);
```

10. It is a compile error to forget the parentheses in the *if* expression.

11. It is a compile error to put a space between the following relational operators: ==, !=, >=, <=. It is also a compile error to reverse them.

12. It is a compile error to use a variable rather than an integral constant as the value in a *case* label.

13. It is a compile error to use the same constant in two *case* labels.

14. The logical operators require two ampersands (&&) or two bars (||). It is a logic error to code them with only one. (Single operators are bitwise operators and are therefore valid code.)

15. It is generally a logic error to use side effects in the second operand in a logical binary expression, as shown below, because the second operand may not be evaluated.

```
(a++ && --b)
```

5.8 Key Terms

break statement
case
case-label
classifying function
comparative operator
complemented *if….else*
conditional expression
converting function
dangling *else*
default label
De Morgan's rule
dependent statement
exclusive *or*
logical operator
multiway selection
negative logic
nested *if* statement
null *else* statement
selection
selection in structure chart
simplifying *if* statements
switch
two-way selection

5.9 Summary

- ❑ Data are called logical if they convey the idea of true or false.
- ❑ C99 implements the logical type, *bool*. It also supports integer logical data: if a data item is nonzero, it is considered *true*; if it is zero, it is considered *false*.
- ❑ C has three operators for combining logical values to create new values: *not*, *and*, *or*.
- ❑ Six comparative operators are used in C: <, <=, >, >=, ==, and !=.
- ❑ Selection in C is done using two statements: *if…else* and *switch*.

❑ The *if...else* construct is used for selection between two alternatives.

❑ You can swap the statements in the true and false branches if you use the complement of an expression in an *if...else* statement.

❑ If the false statement is not required in an *if...else,* it is omitted and the keyword *else* dropped.

❑ If an *else* is dangling, it will be paired with the last unpaired *if.*

❑ Multiway selection can be coded using either the *switch* statement or an *else-if* construct.

❑ The *switch* statement is used to make a decision between many alternatives when the different conditions can be expressed as integral values.

❑ The *else-if* format is used to make multiple decisions when the item being tested is not an integral and therefore a *switch* statement cannot be used.

❑ A labeled statement is used for selection in a *switch* statement.

❑ A *default* label is used as the last label in a *switch* statement, to be executed when none of the *case* alternatives match the tested value.

❑ Indenting the controlled statements in C is good style that enhances the readability of a program.

❑ Selection is used in a structure chart only when it involves a call to another function.

❑ The structure chart for selection shows the paths taken by the logic flow. You cannot always tell by looking at the structure chart which selection will be used (two-way or multiway).

a. A simple *if* is indicated by a diamond below the calling function.

b. An *if...else* and *switch* are indicated by the exclusive or (+).

5.10 Practice Sets

Review Questions

1. Logical data are data that can be interpreted as true or false.

a. True
b. False

2. The expression in a selection statement can have no side effects.

a. True
b. False

3. Each labeled statement may identify one or more statements.

a. True
b. False

4. The character classification functions are found in the standard library (*stdlib.h*).

   a. True
   b. False

5. To ensure that a character is uppercase, the *toupper* conversion function is used.

   a. True
   b. False

6. The ________ logical operator is true only when both operands are true.

   a. and (&&)
   b. greater than (>)
   c. less than (<)
   d. or (||)
   e. not (!)

7. Which of the following is not a comparative operator in C?

   a. <
   b. <=
   c. =
   d. >
   e. >=

8. Two-way selection is implemented with the ________________ statement.

   a. case
   b. else if
   c. switch
   d. the if…else and the switch
   e. if…else

9. Which of the following is not a syntactical rule for the *if…else* statement?

   a. Any expression can be used for the *if* expression.
   b. Only one statement is allowed for the true and the false actions.
   c. The true and the false statements can be another *if…else* statement.
   d. The expression must be enclosed in parentheses.
   e. The selection expression cannot have a side effect.

10. Which of the following statements creates the "dangling *else* problem"?

    a. a nested *if* statement without a false statement
    b. a nested *if* statement without a true statement
    c. a *switch* statement without a default
    d. an *if* statement without a true or a false statement
    e. any nested *if* statement

11. There are two different ways to implement a multiway selection in C. They are

    a. *if...else* and *switch*
    b. *else-if* and *switch*
    c. *if...else* and *else if*
    d. *else-if* and *case*
    e. *switch* and *case*

12. Which of the following statements about *switch* statements is false?

    a. No two *case* labels can have the same value.
    b. The *switch* control expression must be an integral type.
    c. The *case*-labeled constant can be a constant or a variable.
    d. Two *case* labels can be associated with the same statement series.
    e. A *switch* statement can have at most one default statement.

13. Which of the following statements about the *else-if* is false?

    a. Each expression in the *else-if* must test the same variable.
    b. The *else-if* is a coding style rather than a C construct.
    c. The *else-if* requires integral values in its expression.
    d. The *else-if* is used for multiway selections.
    e. The last test in the *else-if* series concludes with a single *else*, which is the default condition.

14. Which of the following is not a character classification in the C language?

    a. ascii
    b. control
    c. digit
    d. graphical
    e. space

Exercises

15. Evaluate the following expressions to *true* or *false*. Show how you arrived at your answer by first adding the default parentheses and then showing the value of each expression as it would be evaluated by C, one expression to a line.

    a. `!(3 + 3 >= 6)`
    b. `1 + 6 == 7 || 3 + 2 == 1`
    c. `1 > 5 || 6 < 50 && 2 < 5`
    d. `14 != 55 && !(13 < 29) || 31 > 52`
    e. `6 < 7 > 5`

16. If x = 0, y = 5, z = 5, what is the value of x, y, and z for each of the following code fragments? (Assume that x, y, and z are their original values for each fragment.)

a.
```
if (z != 0)
    y = 295;
else
x = 10;
```

b.
```
if (y + z > 10)
    y = 99;
    z = 8;
x = ++z;
```

c.
```
if (x == 0)
    {
      x = x - 3;
      z = z + 3;
    }
else
    y = 99;
```

17. If x = 3, y = 0, and z = –4, what is the value of the following expressions?

a. `x && y || z`
b. `x || y && z`
c. `(x && y) || z`
d. `(x || y) && z`
e. `(x && z) || y`

18. Simplify the following expressions by removing the ! operator and the parentheses:

a. `!(x < y)`
b. `!(x >= y)`
c. `!(x == y)`
d. `!(x != y)`
e. `!(!(x > y))`

19. If x = –2, y = 5, z = 0, and t = –4, what is the value of each of the following expressions?

a. `x + y < z + t`
b. `x - 2 * y + y < z * 2 / 3`
c. `3 * y / 4 % 5 && y`
d. `t || z < (y + 5) && y`
e. `!(4 + 5 * y >= z - 4) && (z - 2)`

20. If originally x = 4, y = 0, and z = 2, what is the value of x, y, and z after executing the following code?

```
if (x != 0)
    y = 3;
```

continued

```
else
   z = 2;
```

21. If originally x = 4, y = 0, and z = 2, what is the value of x, y, and z after executing the following code?

```
if (z == 2)
   y = 1;
else
   x = 3;
```

22. If originally x = 4, y = 0, and z = 2, what is the value of x, y, and z after executing the following code?

```
if (x && y)
   x = 3;
else
   y = 2;
```

23. If originally x = 4, y = 0, and z = 2, what is the value of x, y, and z after executing the following code?

```
if (x || y || z)
   y = 1;
else
   z = 3;
```

24. If originally x = 0, y = 0, and z = 1, what is the value of x, y, and z after executing the following code?

```
if (x)
   if (y)
       z = 3;
else
   z = 2;
```

25. If originally x = 4, y = 0, and z = 2, what is the value of x, y, and z after executing the following code?

```
if (z == 0 || x  && !y)
    if (!z)
      y = 1;
else
   x = 2;
```

26. If originally x = 0, y = 0, and z = 1, what is the value of x, y, and z after executing the following code?

```
if (x)
   if (y)
      if (z)
         z = 3;
      else
         z = 2;
```

27. If originally x = 0, y = 0, and z = 1, what is the value of x, y, and z after executing the following code?

```
if (z < x || y >= z && z == 1)
   if (z && y)
       y = 1;
else
   x = 1;
```

28. If originally x = 0, y = 0, and z = 1, what is the value of x, y, and z after executing the following code?

```
if (z = y )
   {
    y++;
    z--;
   }
else
   --x;
```

29. If originally x = 0, y = 0, and z = 1, what is the value of x, y, and z after executing the following code?

```
if (z = x < y)
   {
    x += 3;
    y -= 1;
   }
else
   x = y++;
```

30. If originally x = 0, y = 0, and z = 1, what is the value of x, y, and z after executing the following code?

```
switch (x)
   {
```

continued

```
        case 0   : x = 2;
                   y = 3;
        case 1   : x = 4;
        default  : y = 3;
                   x = 1;
     }
```

31. If originally x = 2, y = 1, and z = 1, what is the value of x, y, and z after executing the following code?

```
switch (x)
    {
     case 0 : x = 2;
              y = 3;
     case 1 : x = 4;
              break;
     default:
              y = 3;
              x = 1;
    }
```

32. If originally x = 1, y = 3, and z = 0, what is the value of x, y, and z after executing the following code?

```
switch ( x )
    {
     case 0 : x = 2;
              y = 3;
              break;
     case 1 : x = 4;
              break;
     default: y = 3;
              x = 1;
    }
```

33. Evaluate the value of the following expressions:

a. `tolower ('C')`
b. `tolower ('?')`
c. `tolower ('c')`
d. `tolower ('5')`

34. Evaluate the value of the following expressions:

a. `toupper ('c')`
b. `toupper ('C')`
c. `toupper ('?')`
d. `toupper ('7')`

Programs

35. Write an *if* statement that will assign the value 1 to the variable `best` if the integer variable `score` is 90 or greater.

36. Repeat Problem 35 using a conditional expression.

37. Write the code to add 4 to an integer variable, `num`, if a float variable, `amount`, is greater than 5.4.

38. Print the value of the integer `num` if the variable `flag` is *true*.

39. Write the code to print either zero or not zero based on the integer variable `num`.

40. If the variable `divisor` is not zero, divide the variable `dividend` by `divisor`, and store the result in `quotient`. If `divisor` is zero, assign it to `quotient`. Then print all three variables. Assume that `dividend` and `divisor` are integers and `quotient` is a double.

41. If the variable `flag` is *true*, read the integer variables `a` and `b`. Then calculate and print the sum and average of both inputs.

42. Rewrite the following code using one *if* statement:

```
if (aChar == 'E')
    c++;
if (aChar == 'E')
    printf ("Value is E\n");
```

43. Rewrite the following code fragment using one *switch* statement:

```
if (ch == 'E' || ch == 'e')
   countE++;
else if (ch == 'A' || ch == 'a')
   countA++
else if (ch == 'I' || ch == 'i')
   countI++;
else
   print ("Error--Not A, E, or I\a\n");
```

44. Write a code fragment that tests the value of an integer `num1`. If the value is 10, square `num1`. If it is 9, read a new value into `num1`. If it is 2 or 3, multiply `num1` by 99 and print out the result. Implement your code using nested *if* statements, not a *switch*.

45. Rewrite Problem 44 using a *switch* statement.

46. Write a code fragment for the flowchart shown in Figure 5-29. Assume that the variables `x` and `y` are integers and `z` is a float-point number.

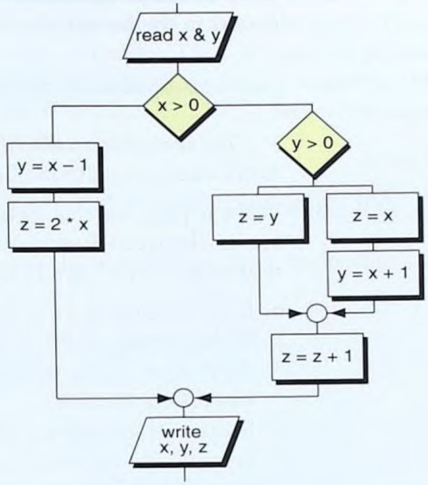


FIGURE 5-29 Flowchart for Problem 46

47. Write a function called `smallest` that, given three integers, returns the smallest value.

48. Write a function called `day_of_week` that, given an integer between 0 and 6, prints the corresponding day of the week. Assume that the first day of the week (0) is Sunday.

49. Write a function called `month_of_year` that, given an integer between 1 and 12, prints the corresponding month of the year.

50. Write a function called `parkingCharge` that, given the type of vehicle (`c` for car, `b` for bus, `t` for truck) and the hours a vehicle spent in the parking lot, returns the parking charge based on the rates shown below.

`car`	`$2 per hour`
`bus`	`$3 per hour`
`truck`	`$4 per hour`

51. Write a function to find the smallest of four integers.

Projects

52. Complete the incremental implementation of Program 5-13. First write the code to implement the multiply function, then the divide function, and finally the print results function. The print results function should

use a *switch* to determine which option was requested and then print the results in the format shown below.

```
number operator number = result
```

You may pattern the function after the print results in Chapter 4, but it will need extensive modification.

53. Write a program that determines a student's grade. It reads three test scores (between 0 and 100) and calls a function that calculates and returns a student's grade based on the following rules:

a. If the average score is 90% or more, the grade is `A`.

b. If the average score is 70% or more and less than 90%, it checks the third score. If the third score is more than 90%, the grade is `A`; otherwise, the grade is `B`.

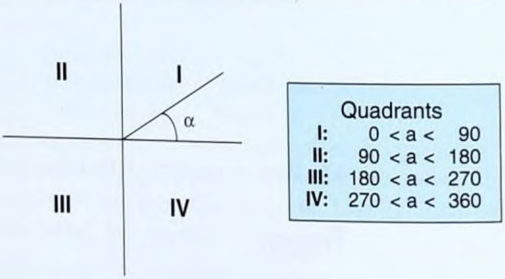
c. If the average score is 50% or more and less than 70%, it checks the average of the second and third scores. If the average of the two is greater than 70%, the grade is `C`; otherwise, it is `D`.

d. If the average score is less than 50 percent, then the grade is `F`.

The program's *main* is to contain only call statements. At least three subfunctions are required: one to read scores, one to determine the grade, and one to print the results.

54. In Program 4-7, "Strange College Fees," we wrote a program to calculate college fees. Modify this program for Typical College. At Typical College, the students pay a fee of $10 per unit for up to 12 units; once they have paid for 12 units, they have no additional per-unit fee. The registration fee remains $10 but is assessed only if courses are taken in the term.

55. Given a point, a line from the point forms an angle with the horizontal axis to the right of the line. The line is said to terminate in one of four quadrants based on its angle (`a`) from the horizontal, as shown in Figure 5-30.



Quadrants	
I:	0 < a < 90
II:	90 < a < 180
III:	180 < a < 270
IV:	270 < a < 360

FIGURE 5-30 Quadrants for Project 55

Write a program that determines the quadrant, given a user-input angle. Use a function to read and validate the angle. Note: If the angle is exactly 0°, it is not in a quadrant but lies on the positive X-axis; if it is exactly 90°, it lies on the positive Y-axis; if it is exactly 180°, it lies on the negative X-axis; and if it is exactly 270°, it lies on the negative Y-axis. Test your program with the following data:

0°, 48.3°, 90°, 179.8°, 180°, 186°, 270°, 300°, and 360°

56. How many values of the variable num must be used to completely test all branches of the following code fragment?

```
if (num > 0)
   if (value < 25)
      {
       value = 10 * num;
       if (num < 12)
           value = value / 10;
      } // if value
   else
      value = 20 * num;
else
   value = 30 * num;
```

57. Write a program that asks the user to enter the current date and a person's birth date in the form month, day, year. The program then calculates the person's age in integral years. Use separate functions to enter the dates (pass by address), calculate the person's age, and print the results. Test your program with the following dates: 11/14/1957, 5/10/1989, and 1/5/2000.

58. Write a C program to calculate the parking fare for customers who park their cars in a parking lot when the following information is given:

   a. A character showing the type of vehicle: `C` for car, `B` for bus, `T` for truck.
   b. An integer between 0 and 24 showing the hour the vehicle entered the lot.
   c. An integer between 0 and 60 showing the minute the vehicle entered the lot.
   d. An integer between 0 and 24 showing the hour the vehicle left the lot.
   e. An integer between 0 and 60 showing the minute the vehicle left the lot.

   This is a public lot. To encourage people to park for a short period of time, the management uses two different rates for each type of vehicle, as shown in Table 5-11.

Vehicle	First Rate	Second Rate
CAR	$0.00/hr first 3 hr	$1.50/hr after 3 hr
TRUCK	$1.00/hr first 2 hr	$2.30/hr after 2 hr
BUS	$2.00/hr for first hr	$3.70/hr after first hr

TABLE 5-11 Rates for Project 58

No vehicle is allowed to stay in the parking lot later than midnight; it will be towed away.

The input data consist of a character and a set of four integers representing the type of vehicle and the entering and leaving hours and minutes. But these pieces of data must be input into the computer in a user-friendly way. In other words, the computer must prompt the user to enter each piece of data as shown below. (Color indicates typical data.)

```
Type of vehicle? C
Hour vehicle entered lot   (0 - 24)?  14
Minute vehicle entered lot (0 - 60)?  23
Hour vehicle left lot      (0 - 24 )? 18
Minute vehicle left lot    (0 - 60)?  8
```

The output format is shown below.

```
             PARKING LOT CHARGE

Type of vehicle: Car or Bus or Truck
TIME-IN              XX : XX
TIME-OUT             XX : XX
                    --------
PARKING TIME           XX:XX
ROUNDED TOTAL             XX
                    --------
TOTAL CHARGE          $XX.XX
```

This program must first calculate the actual time spent in the parking lot for each vehicle. This means using modulo arithmetic to handle time calculation. We can calculate this in many ways, one of which is shown below. To calculate the time spent in the parking lot, use the following algorithm:

a. Compare the minute portion of the leaving and the entering time.

If the first one is smaller than the second,

- ❑ Add 60 to the minute portion of the leaving time.
- ❑ Subtract 1 from the hour portion of the leaving time.

b. Subtract the hour portions.

c. Subtract the minute portions.

d. Since there are no fractional hour charges, the program must also round the parking time up to the next hour before calculating the charge. The program should use the *switch* statement to distinguish between the different types of vehicles.

A well-structured program design is required. A typical solution will use several functions besides *main*. Before you start programming, prepare a structure chart. Run your program six times with the data shown in Table 5-12.

Test	Type	Hour In	Minute In	Hour Out	Minute Out
1	C	12	40	14	22
2	B	8	20	8	40
3	T	2	0	3	59
4	C	12	40	16	22
5	B	8	20	14	20
6	T	2	0	12	0

TABLE 5-12 Test Data for Project 58

59. This program is a simple guessing game. The computer is to generate a random number between 1 and 20. The user is given up to five tries to guess the exact number. After each guess, you are to tell the user if the guessed number is greater than, less than, or equal to the random number. If it is equal, no more guesses should be made. If the user hasn't guessed the number after five tries, display the number with a message that the user should know it by now and terminate the game.

A possible successful dialog:

```
I am thinking of a number between 1 and 20.
Can you guess what it is?        10
Your guess is low. Try again:    15
Your guess is low. Try again:    17
Your guess is high. Try again:   16

Congratulations! You did it.
```

A possible unsuccessful dialog:

```
I am thinking of a number between 1 and 20.
Can you guess what it is?        10
```

continued

```
Your guess is low. Try again:   20
Your guess is high. Try again:  10
Your guess is low. Try again:   18
Your guess is high. Try again:  12

Sorry. The number was 15.
You should have gotten it by now.
Better luck next time.
```

Your design for this program should include a separate function to get the user's guess, a function to print the unsuccessful message, one to print the successful message, and one to print the sorry message.

60. Write a program that, given a person's birth date (or any other date in the Gregorian calendar), will display the day of the week the person was born.

To determine the day of the week, you will first need to calculate the day of the week for December 31 of the previous year. To calculate the day for December 31, use the following formula.

$$\left((\mathbf{year}-1)\times 365+\left\lceil\frac{\mathbf{year}-1}{4}\right\rceil-\left\lceil\frac{\mathbf{year}-1}{100}\right\rceil+\left\lceil\frac{\mathbf{year}-1}{400}\right\rceil\right)\ \%\ 7$$

The formula determines the day based on the values as shown below.

Day 0: Sunday
Day 1: Monday
Day 2: Tuesday
Day 3: Wednesday
Day 4: Thursday
Day 5: Friday
Day 6: Saturday

Once you know the day for December 31, you simply calculate the days in the year before the month in question. Use a *switch* statement to make this calculation. (Hint: Use *case* 12 first, and then fall into *case* 11, 10...2.) If the desired month is 12, add the number of days for November (30). If it is 11, add the number of days for October (31). If it is 3, add the number of days for February (28). If it is 2, add the number of days for January (31). If you do not use a *break* between the months, the *switch* will add the days in each month before the current month.

To this figure, add the day in the current month and then add the result to the day code for December 31. This number modulo seven is the day of the week.

There is one more refinement. If the current year is a leap year, and if the desired date is after February, you need to add 1 to the day code. The following formula can be used to determine if the year is a leap year.

```
(!(year % 4) && (year % 100)) || !(year % 400)
```

Your program should have a function to get data from the user, another to calculate the day of the week, and a third to print the result.

To test your program, run it with the following dates:

a. February 28, 1900, and March 1, 1900

b. February 28, 1955, and March 1, 1955

c. February 28, 1996, and March 1, 1996

d. February 28, 2000, and March 1, 2000

e. December 31, 1996

f. The first and last dates of the current week.

61. Write a program that calculates the change due a customer by denomination; that is, how many pennies, nickels, dimes, etc. are needed in change. The input is to be the purchase price and the size of the bill tendered by the customer ($100, $50, $20, $10, $5, $1).

62. Write a menu-driven program that allows a user to enter five numbers and then choose between finding the smallest, largest, sum, or average. The menu and all the choices are to be functions. Use a *switch* statement to determine what action to take. Provide an error message if an invalid choice is entered.

Run the program five times, once with each option and once with an invalid option. Each run is to use the following set of data:

18, 21, 7, 54, 9

63. Write a program that tests a user-entered character and displays its classification according to the ASCII classifications shown in Figure 5-25, "Classifications of the Character Type." Write the program starting at the top of the classification tree and display all classifications that the character belongs to. For instance, if the user enters a digit, you should display that it is printable, graphical, alphanumeric, and a digit.

64. Write a program to compute the real roots of a quadratic equation ($ax^2 + bx + c = 0$). The roots can be calculated using the following formulas:

$$x1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad x2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Your program is to prompt the user to enter the constants (a, b, c). It is then to display the roots based on the following rules:

a. If both a and b are zero, there is no solution.

b. If a is zero, there is only one root ($-c$ / b).

c. If the discriminate ($b^2 - 4ac$) is negative, there are no real roots.

d. For all other combinations, there are two roots.

Test your program with the data in Table 5-13.

a	b	c
3	8	5
–6	7	8
0	9	–10
0	0	11

TABLE 5-13 Test Data for Project 64

Chapter 6

Repetition

The real power of computers is in their ability to repeat an operation or a series of operations many times. This repetition, called looping, is one of the basic structured programming concepts. In this chapter, we discuss looping and introduce different looping constructs. First we define the basic concepts of loops, including a most important concept: how to stop a loop. We then present the three different loop constructs and take you through the C implementation of these three constructs. In this discussion, we include some basic loop applications. After discussing the basic loop constructs, we introduce an implicit loop construct, recursion. We conclude with a discussion of some of the software engineering implications of loops.

Objectives

- ❑ To understand basic loop concepts:
    - ■ pretest loops and post-test loops
    - ■ loop initialization and updating
    - ■ event and counter-controlled loops
- ❑ To understand and be able to select the best loop construct for a given problem.
- ❑ To write programs that use the *while*, *for*, or *do...while* statements.
- ❑ To understand the basic concepts and usage of recursion algorithms.
- ❑ To understand and be able to determine the efficiency of an algorithm through an analysis of its looping constructs.

6.1 Concept of a loop

The concept of a loop is shown in the flowchart segment in Figure 6-1. In this flowchart, the action is repeated over and over again. It never stops.

Since the loop in Figure 6-1 never stops, it is the computer version of the perpetual motion machine. The action (or actions) will be repeated forever. We definitely do not want this to happen. We want our loop to end when the work is done. To make sure that it ends, we must have a condition that controls the loop. In other words, we must design the loop so that before or after each **iteration**, it checks to see if the task is done. If it is not done, the loop repeats one more time; if the task is done, the loop terminates. This test is known as a **loop control expression**.

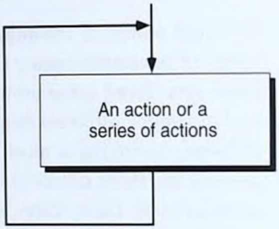


FIGURE 6-1 Concept of a Loop

6.2 Pretest and Post-test Loops

We have established that we need to test for the end of a loop, but where should we check it—before or after each iteration? Programming languages allow us to check the loop control expression either before *or* after each iteration of the loop. In other words, we can have either a pre- or a post-test terminating condition. In a **pretest loop**, the condition is checked before we start and at the beginning of each iteration. If the test condition is true, we execute the code; if the test condition is false, we terminate the loop.

Pretest Loop

In each iteration, the control expression is tested first. If it is true, the loop continues; otherwise, the loop is terminated.

Post-test Loop

In each iteration, the loop action(s) are executed. Then the control expression is tested. If it is true, a new iteration is started; otherwise, the loop terminates.

In the **post-test loop**, we always execute the action at least once. The loop control expression is then tested. If the expression is true, the loop repeats; if the expression is false, the loop terminates. The flowcharts in Figure 6-2 show these two loop types.

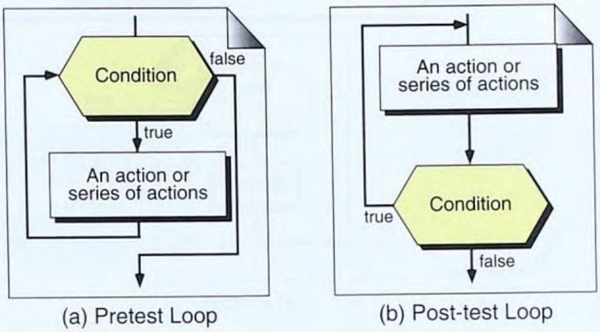


FIGURE 6-2 Pretest and Post-test Loops

Consider an example of pretest and post-test looping. Imagine that you are ready to start your daily exercises. Your exercise program requires you to do as many push-ups as possible. You can check your limit using either a pretest or a post-test condition. In the pretest strategy, you first check to see if you have enough energy to start. In the post-test strategy, you do one push-up and then you test to see if you have enough energy to do another one. Note that in both cases the question is phrased so that if the answer is true you continue the loop. The two strategies are shown in Figure 6-3.

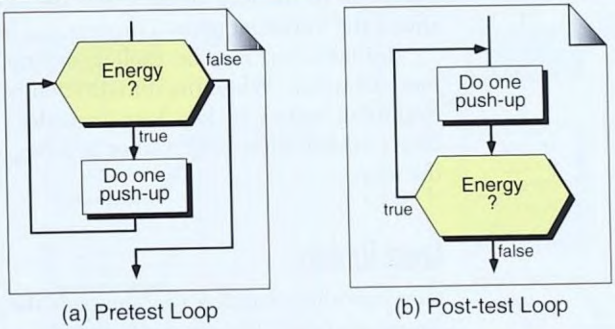


FIGURE 6-3 Two Different Strategies for Doing Exercises

As you can see, in the first strategy, you may not do any push-ups. If you are tired when you start and don't have the energy for at least one push-up, you are done. In the second strategy, you must do at least one push-up. In other words, in a pretest loop, the action may be done zero, one, or more times; in a post-test loop, the action is done one or more times. This major difference between a pretest and a post-test loop, which you must clearly understand, is shown in Figure 6-4.

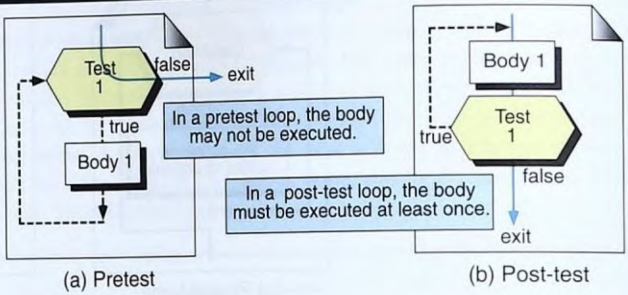

Test 1
false
exit
true
Body 1
In a pretest loop, the body may not be executed.
(a) Pretest
Body 1
true
Test 1
false
In a post-test loop, the body must be executed at least once.
exit
(b) Post-test

FIGURE 6-4 Minimum Number of Iterations in Two Loops

6.3 Initialization and Updating

In addition to the loop control expression, two other processes, initialization and updating, are associated with almost all loops.

Loop Initialization

Before a loop can start, some preparation is usually required. We call this preparation loop **initialization**. Initialization must be done before the first execution of the loop body. It sets the stage for the loop actions. Figure 6-5 shows the initialization as a process box before the loop.

Initialization may be explicit or implicit. Explicit initialization is much more common. When the initialization is explicit, we include code to set the beginning values of key loop variables. Implicit initialization provides no direct initialization code; rather, it relies on a preexisting situation to control the loop.

Loop Update

How can the condition that controls the loop be true for a while and then change to false? The answer is that something must happen inside the body of the loop to change the condition. Otherwise, we would have an **infinite loop**. For example, in the loops shown in Figure 6-3, you gradually lose your energy until a point comes when you cannot continue the push-ups. This changes the resulting condition from true to false. The actions that cause these changes are known as **loop update**. Updating is done in each iteration, usually as the last action. If the body of the loop is repeated *m* times, then the updating is also done *m* times.

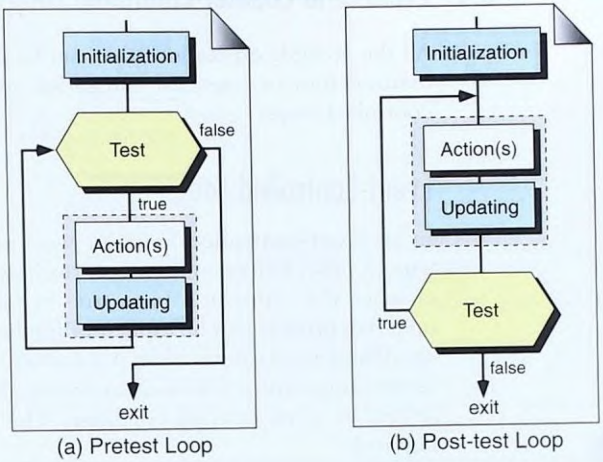


FIGURE 6-5 Loop Initialization and Updating

The concepts of initialization and updating can be applied to our previous push-up example. In this case, initialization is created by nutrition, an implicit initialization. During each push-up, some of the initial energy is consumed in the process and your energy is reduced, which updates your energy level. This is shown in Figure 6-6.

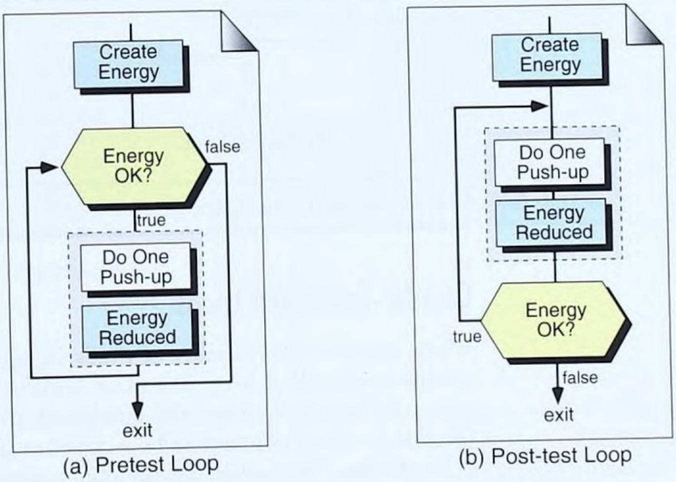


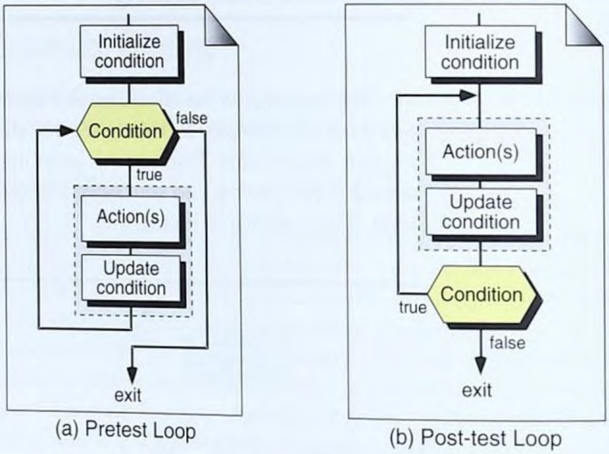
FIGURE 6-6 Initialization and Updating for Exercise

6.4 Event- and Counter-Controlled Loops

All the possible expressions that can be used in a loop limit test can be summarized into two general categories: event-controlled loops and counter-controlled loops.

Event-Controlled Loops

In an **event-controlled loop**, an event changes the control expression from true to false. For example, when reading data, reaching the end of the data changes the expression from true to false. In event-controlled loops, the updating process can be explicit or implicit. If it is explicit, such as finding a specific piece of information, it is controlled by the loop. If it is implicit, such as the temperature of a batch of chemicals reaching a certain point, it is controlled by some external condition. The event-controlled loop is shown in Figure 6-7.



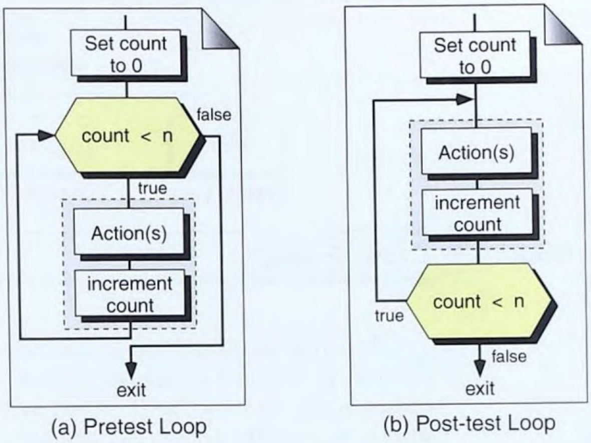
(a) Pretest Loop

(b) Post-test Loop

FIGURE 6-7 Event-controlled Loop Concept

Counter-Controlled Loops

When we know the number of times an action is to be repeated, we use a **counter-controlled loop**. We must initialize, update, and test the counter. Although we need to know the number of times we want to execute the loop, the number does not need to be a constant. It can also be a variable or a calculated value. The update can be an increment, in which case we are counting up, or a decrement, in which case we are counting down. The counter-controlled loop is shown in Figure 6-8.



(a) Pretest Loop

(b) Post-test Loop

FIGURE 6-8 Counter-controlled Loop Concept

Loop Comparison

The number of iterations of a loop is given as *n*. In a pretest loop, when we come out of the loop, the limit test has been executed *n* + 1 times. In a post-test loop, when we come out of the loop, the limit test has been executed only *n* times. A summary of the two different loop concepts is shown in Table 6-1.

Pretest Loop		**Post-test Loop**	
Initialization:	1	Initialization:	1
Number of tests:	*n* + 1	Number of tests:	n
Action executed:	n	Action executed:	n
Updating executed:	n	Updating executed:	n
Minimum iterations:	0	Minimum iterations:	1

TABLE 6-1 Loop Comparisons

6.5 Loops in C

C has three loop statements: the *while,* the *for,* and the *do...while*. The first two are pretest loops, and the *do...while* is a post-test loop. We can use all of them for event-controlled and counter-controlled loops. The *while* and *do...while* are most commonly used for event-controlled loops, and the *for* is usually used for counter-controlled loops. These loop constructs are shown in Figure 6-9.

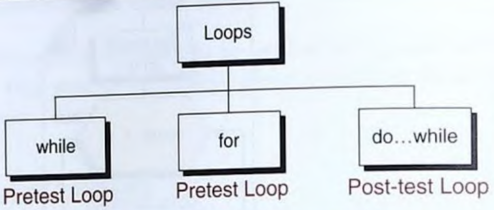
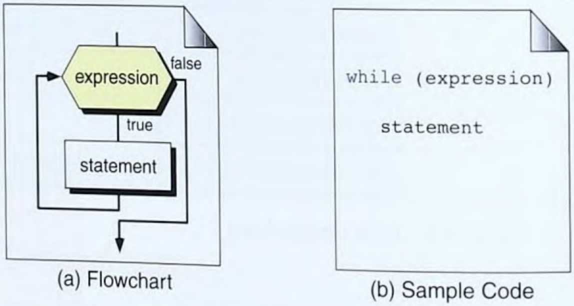


FIGURE 6-9 C Loop Constructs

Note that all three of these loop constructs continue when the limit control test is true and terminate when it is false. This consistency of design makes it easy to write the limit test in C. On the other hand, general algorithms are usually written just the opposite because analysts tend to think about what will terminate the loop rather than what will continue it. Therefore, we must complement or otherwise modify the limit test when we write our program. This is one place where "De Morgan's Rule," discussed in Chapter 5, is very handy.

The *while* Loop

The ***while*** loop is a pretest loop. It uses an expression to control the loop. Because it is a pretest loop, it tests the expression before every iteration of the loop. The basic syntax of the *while* statement is shown in Figure 6-10. No semicolon is needed at the end of the *while* statement. When you see a semicolon, it actually belongs to the statement within the *while* statement.



(a) Flowchart

```
while (expression)

   statement
```

(b) Sample Code

FIGURE 6-10 The *while* Statement

Note that the sample code in Figure 6-10 shows that the loop body is a single statement; that is, the body of the loop must be one, and only one, statement. If we want to include multiple statements in the body, we must put them in a compound statement (block). This concept is shown in Figure 6-11.

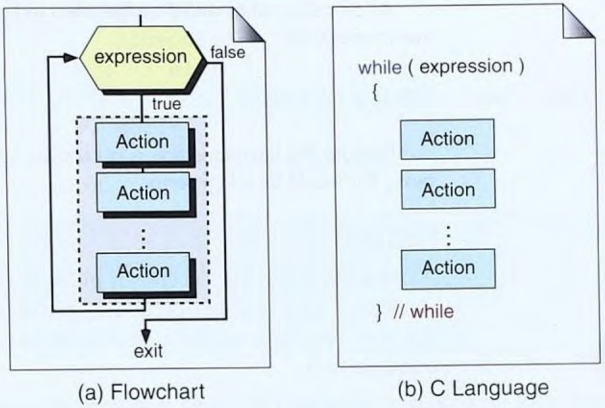


FIGURE 6-11 Compound *while* Statement

Process-Control Loops

Perhaps the simplest loop is the loop that never ends. While it is virtually never used in data processing, it is common in **process-control loops** such as network servers, environmental systems, and manufacturing systems. A simple process-control system that might be used to control the temperature in a building is shown in Program 6-1.

PROGRAM 6-1 Process-control System Example

```
while (true)
   {
    temp = getTemperature();
    if (temp < 68)
        turnOnHeater(); turnOffAirCond()
    else if (temp > 78)
        turnOnAirCond(); turnOffHeater()
    else
        {
         turnOffHeater();
         turnOffAirCond();
        } // else
   } // while true
```

Program 6-1 Analysis The limit test in this simple program is a literal constant. It cannot be changed. Therefore, no update is required. Because the limit test in the *while* statement is always true, it will never stop. We must emphasize, however, that this is not a good construct for anything other than a loop that truly never ends.

As an aside, what would be the effect in Program 6-1 if the following *while* statement were used?

```
while (0)
```

Because the limit condition is a constant false, the loop would never start. Obviously, this would be a logic error.

Print Loops

A common *while* loop is shown in Program 6-2. In this case we want to print a series of numbers in descending order. To keep from running off the end of the line, we have added a test to write a newline when ten numbers have been written.

PROGRAM 6-2 A *while* Loop to Print Numbers

```
/* Simple while loop that prints numbers 10 per line.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int num;
   int lineCount;

// Statements
   printf ("Enter an integer between 1 and 100: ");
   scanf  ("%d", &num);         // Initialization

   // Test number
   if (num > 100)
       num = 100;

   lineCount = 0;
   while (num > 0)
      {
       if (lineCount < 10)
           lineCount++;
```

continued

PROGRAM 6-2 A *while* Loop to Print Numbers *(continued)*

```
26          else
27            {
28             printf("\n");
29             lineCount = 1;
30            } // else
31         printf("%4d", num--);          // num-- updates loop
32        } // while
33      return 0;
34   }  // main
```

```
Results:
Enter an integer between 1 and 100: 15
   15  14  13  12  11  10   9   8   7   6
    5   4   3   2   1
```

Program 6-2 Analysis Find the basic elements of a loop in this program. First, look for loop initialization, then the limit test, and finally the update.

The initialization is done by asking the user to enter num in Statement 14. Note how we make sure that the user enters a number that is not too large. If the number is over 100, we simply set it to 100, which is the maximum we told the user to enter. The limit test is in Statement 22. As long as num is greater than 0, we continue printing out the number series. The update is hidden in Statement 31; after printing num, we subtract 1 from it. Although this is a common C programmer trick, the program would have been easier to read if we had put the update (num--) on the line after the print. That way it would have been obvious.

File Loops

One of the most common loop applications in any language is reading data until all the data have been processed—that is, until the end of file is reached. In C, the *scanf* function returns the system constant EOF when it detects an end-of-file.

Consider this application of the *scanf* feature: suppose we want to read and process a list of numbers from the keyboard. We type all the numbers, each one on a separate line. At the end, we can type end-of-file (<ctrl+d> in UNIX or <ctrl+z> in DOS). This loop logic is shown in the next example.

```
ioResult = scanf ("%d", &a);
while (ioResult != EOF)
   {
    // Action: Process data
   …
   ioResult = scanf ("%d", &a);
  }  // while
```

But no one codes a *while* loop like this. Let's simplify it. In the revised version of the loop, we have moved the *scanf* to the conditional expression in the *while* statement itself. We can do this because the initialization and the update are identical. The initialization and updating are now both self-contained parts of the *while* statement. This change is shown below.

```
while ((ioResult = scanf ("%d", &a)) != EOF)
   {
    // Process data
    …
   } // while
```

But we can simplify this loop even further. Because the *scanf* function returns a value, we can test for end-of-file in the *while* expression. We don't need the variable, `ioResult`, so we can simply use the function value and discard it after it has been checked. The result is the standard C loop for reading and processing data from a file shown in Program 6-3, which adds a list of integers read from the keyboard and displays their sum.

PROGRAM 6-3 Adding a List of Numbers

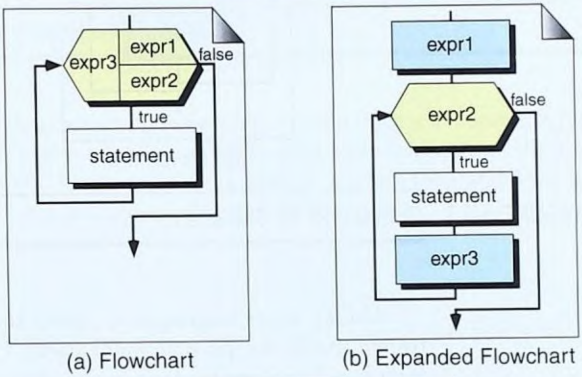
```
/* Add a list of integers from the standard input unit
   Written by:
   Date:
*/
#include <stdio.h>
int main (void)
{
// Local Declarations
   int x;
   int sum = 0;

// Statements
   printf("Enter your numbers: <EOF> to stop.\n");
   while (scanf("%d", &x) != EOF)
      sum += x;
   printf ("\nThe total is: %d\n", sum);
   return  0;
} // main
```

```
Results:
Enter your numbers: <EOF> to stop.
15
22
3^d
The total is: 40
```

Program 6-3 Analysis Note that a compound statement (block) is not needed in the *while* loop, because the addition can be done in one statement. Another important point is that the statement to print the sum is outside the loop. Since the user can see the input values on the screen, all we need to show is the sum.

The *for* Loop

The *for* statement is a pretest loop that uses three expressions. The first expression contains any initialization statements, the second contains the **limit-test expression**, and the third contains the updating expression. Figure 6-12 shows a flowchart, and an expanded interpretation, for a sample *for* statement.



(a) Flowchart

(b) Expanded Flowchart

```
for (expr1; expr2; expr3)
     statement
```

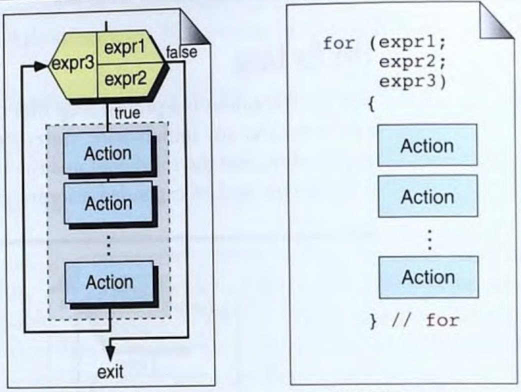
FIGURE 6-12 *for* Statement

Expression 1 is executed when the *for* starts. Expression 2 is the limit test expression. As shown in the expanded flowchart, it is tested *before* every iteration. Remember that since the *for* is a pretest loop, the body is not executed if the limit condition is false at the start of the loop.

Finally, Expression 3 is the update expression. It is executed at the end of each loop. Note that the code in the *for* statement must be expressions. This means that you cannot use statements, such as *return*, in the *for* statement itself. Like the *while* statement, the *for* statement does not need a semicolon.

C allows the limit test in Expression 2 to be a variable. It also allows us to change the value of the variable during the execution of the loop. We do not recommend changing it in the loop, however. It is not a good structured programming style and can lead to errors and perpetual loops.

The **body** of the *for* loop must be one, and only one, statement. If we want to include more than one statement in the body, we must code them in a compound statement. A *for* statement with a compound statement is shown in Figure 6-13.



```
for (expr1;
     expr2;
     expr3)
  {
   Action
   Action
     ⋮
   Action
  } // for
```

(a) Flowchart

(b) C Language

FIGURE 6-13 Compound *for* Statement

Unlike some languages, C allows the loop control expression to be controlled inside the *for* statement itself. This means that the updating of the limit condition can be done in the body of the *for* statement. In fact, Expression 3 can be null and the updating controlled entirely within the body of the loop. This is not a recommended structured programming coding technique, although it is required in some situations.

Although the *for* loop can be used anywhere a pretest loop is required, it is most naturally used for counter-controlled loops (see "Counter-Controlled Loops" in Section 6.4). Its self-contained design makes it ideal for count logic.

> A *for* loop is used when a loop is to be executed a known number of times. We can do the same thing with a *while* loop, but the *for* loop is easier to read and more natural for counting loops.

Let's compare the *while* and the *for* loops. Figure 6-14 shows a graphical representation of each side by side. Note that the *for* loop contains the initialization, update code, and limit test in one statement. This makes for very readable code. All the control steps, initialization, end-of-loop test, and updating are done in one place. This is a variation of the structured programming concepts of encapsulation, in which all code for a process is placed in one module. Another way of looking at it is that a *for* loop communicates better and is more compact.

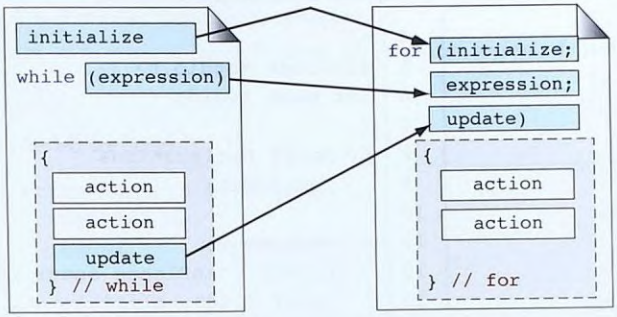


FIGURE 6-14 Comparing *for* and *while* Loops

Now let's solve the same problem using both a *while* and a *for* loop. The code shown below contains a loop to read 20 numbers from the keyboard and find their sum. This can be done both in a *while* loop and a *for* loop. But as you can see, the *for* loop is more self-documenting.

```
i   = 1;
sum = 0;
while (i <= 20)
   {
    scanf("%d", &a);
    sum += a;
    i ++;
   } // while
```

```
sum = 0;
for (i = 1; i <= 20; i++)
     {
      scanf ("%d", &a);
      sum += a;
     } // for
```

C99 modified the *for* syntax to allow the definition of loop variables to be done in the *for* statement itself. This syntax is shown in the next example. Note, however, that definitions of the *for* statement are local to the loop body; they are not in scope when the loop terminates. If we need them after the loop, they must be defined in the local declarations section.

Example: Print Number Series

To demonstrate the *for* loop, let's write a program that asks the user for a number and then prints the series of numbers starting at one and continuing up to and including the user-entered number. The code is in Program 6-4.

PROGRAM 6-4 Example of a *for* Loop

```
/* Print number series from 1 to user-specified limit.
      Written by:
      Date:
```

continued

PROGRAM 6-4 Example of a *for* Loop *(continued)*

```
*/
#include <stdio.h>
int main (void)
{
// Local Declarations
   int limit;

// Statements
   printf ("\nPlease enter the limit: ");
   scanf ("%d", &limit);
   for (int i = 1; i <= limit; i++)
      printf("\t%d\n", i);
   return 0;
}  // main
```

```
Results:
Please enter the limit: 3
  1
  2
  3
```

Program 6-4 Analysis This simple program is the model for many looping functions. Let's look at three simple modifications to it. First, how would you print only odd numbers? This requires a change only to the update in the *for* statement.

```
for (i = 1; i <= limit; i += 2)
```

Now let's change it to print the numbers backward. In this case, all the statements in the *for* statement must be changed, but the rest of the program is still unchanged.

```
for (i = limit; i >= 1; i--)
```

For the final example, let's print the numbers in two columns, the odd numbers in the first column and the even numbers in the second column. We must modify the update statement in the *for* statement and also the print statement as shown below.

```
for (i = 1; i <= limit; i += 2)
    printf("\t%2d\t%2d\n", i, i + 1);
```

We have used tabs and width specifications to align the output in columns. Note that the second print value (`i + 1`) is an expression. It does *not* change the value of `i`. There is no side effect, it is just a value.

Nested *for* Loops

Any statement, even another *for* loop, can be included in the body of a *for* statement. Using nested loops can create some interesting applications. Let's look at a very simple one here. We will give other examples in Section 6.8 "Looping Applications," later in the chapter.

Program 6-5 uses a nested loop to print a series of numbers on multiple lines.

PROGRAM 6-5 A Simple Nested *for* Loop

```
/* Print numbers on a line.
    Written by:
    Date:
*/
#include <stdio.h>

int main (void)
{
// Statements
  for (int i = 1; i <= 3; i++)
    {
     printf("Row %d: ", i);
     for (int j = 1; j<= 5; j++)
        printf("%3d", j);
     printf("\n");
    } // for i
  return 0;
}  // main
```

```
Results:
Row 1:    1  2  3  4  5
Row 2:    1  2  3  4  5
Row 3:    1  2  3  4  5
```

The *do...while* Loop

The *do...while* statement is a post-test loop. Like the *while* and *for* loops, it also uses an expression to control the loop, but it tests this expression *after* the execution of the body. The format of the *do...while* statement is shown in Figure 6-15.

The body of the *do...while* loop must be one, and only one, statement. If we need to include multiple statements in the body, we must put them in a compound statement. The second example in Figure 6-15 shows the logic flow and code for a *do...while* statement that uses a compound statement. Look carefully at the code block in the second example. Note that the *do* and the *while* braces are aligned. Note also that the *while* expression follows the brace on the same line. This is a good style because it makes it easy for the reader to see the statement. Finally, note that the *do...while* is concluded with a semicolon. This differs from the other looping constructs that we have seen.

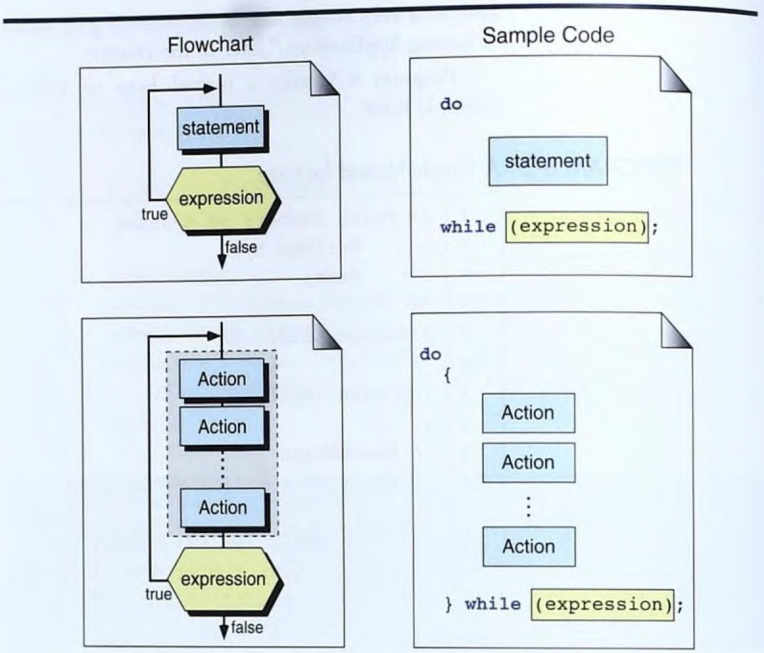


FIGURE 6-15 *do…while* Statement

Let's look at a simple program that uses a loop to print five numbers. We code two loops, first a *while* loop and then a *do…while*. Although one loop is a pretest and the other is a post-test, both print the same series. The code is seen in Program 6-6.

PROGRAM 6-6 Two Simple Loops

```
/* Demonstrate while and do...while loops.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int loopCount;

// Statements
```

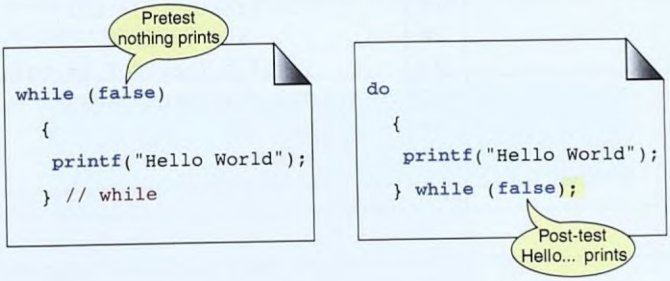
continued

PROGRAM 6-6 Two Simple Loops *(continued)*

```
   loopCount = 5;
   printf("while loop      : ");
   while (loopCount > 0)
      printf ("%3d", loopCount--);
   printf("\n\n");

   loopCount = 5;
   printf("do...while loop: ");
   do
      printf ("%3d", loopCount--);
   while (loopCount > 0);
   printf("\n");
   return 0;
}  // main
```

```
Results
while loop      :   5  4  3  2  1

do...while loop:   5  4  3  2  1
```

Program 6-6 Analysis Because the *do...while* limit test isn't done until the end of the loop, we use it when we know that the body of the loop must be done at least once. To demonstrate the impact of the two loops, study the code in Figure 6-16. In the *while* loop, the message is not printed, because the limit condition is tested first. In the *do...while* loop, even though the limit test is false, the message is printed because the message is printed before the limit test.



```
while (false)
  {
   printf("Hello World");
  } // while
```

```
do
  {
   printf("Hello World");
  } while (false);
```

FIGURE 6-16 Pre- and Post-test Loops

Programmers commonly use the *do...while* loop in data validation to make a program robust. For example, consider an application that requires that we read an integer between 10 and 20. We can use the *do...while* loop as shown in the next example.

```
do
   {
    printf ("Enter a number between 10 & 20: ");
    scanf ("%d", &a);
   } while (a < 10 || a > 20);
```

To demonstrate the *do...while*, let's rewrite Program 6-3, "Adding a List of Numbers." The modified code is shown in Program 6-7.

PROGRAM 6-7 Adding a List with the *do...while*

```
/* Add a list of integers from the standard input unit
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int x;
   int sum = 0;
   int testEOF;

   // Statements
   printf("Enter your numbers: <EOF> to stop.\n");
   do
       {
        testEOF = scanf("%d", &x);
        if (testEOF != EOF)
            sum += x;
       } while (testEOF != EOF);
   printf ("\nTotal: %d\n", sum);
   return  0;
}  // main
```

```
Results:
Run 1:
   Enter your numbers: <EOF> to stop.
   10 15 20 25 ^d
   Total: 70

Run 2:
   Enter your numbers: <EOF> to stop.
   ^d
   Total: 0
```

Program 6-7 Analysis Since the *do...while* always executes the body of the loop at least once, we had to make some changes. Compare Program 6-3 and Program 6-7 carefully. Note that the *scanf* is no longer in the loop limit test. Since the limit test is after the loop body, the *scanf* must be moved to the beginning of the loop. The result of the input is saved in a new variable, `testEOF`. Finally, before we can add the value we read to the accumulator, we must ensure we are not at the end of the file. The add statement is therefore guarded by an *if* statement. Although Program 6-7 is a little less efficient than Program 6-3, it does the same job.

The Comma Expression

A **comma expression** is a complex expression made up of two expressions separated by a comma. Although it can legally be used in many places, it is most often used in *for* statements. The expressions are evaluated left to right. The value and type of the expressions are the value and type of the right expression; the left expression is included only for its side effect. The comma operator has the lowest priority of all operators, priority 1.

The following statement is a modification of the *for* statement code shown on page 317. It uses a comma expression to initialize the accumulator, `sum`, and the index, `i`, in the loop. In this example, the value of the comma expression is discarded. This is a common use of the **comma operator.**

```
for (sum = 0, i = 1; i <= 20; i++)
    {
     scanf("%d", &a);
     sum += a;
    } // for
```

Comma expressions can be nested. When they are nested, all expression values other than the last are discarded. Figure 6-17 shows the format of a nested comma expression.

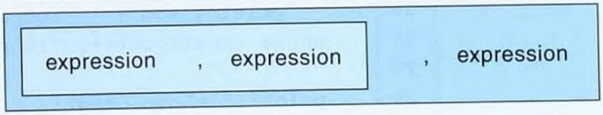


FIGURE 6-17 Nested Comma Expression

A final word of caution. Remember that the value of the expression is the value of the rightmost expression. Although it is not recommended, if you use a comma expression for the second expression in a *for* loop, make sure that the loop control is the last expression.

Let's use the comma expression to demonstrate the difference between the *while* and the *do...while*. As shown in Table 6-1, "Loop Comparisons," the only difference is the number of limit tests that are made. Program 6-6 shows that the same job can be done by either loop. Program 6-8 uses both loops to count from 1 to 10. It uses the comma expression to count the number of limit tests in each loop.

PROGRAM 6-8 Comparison of *while* and *do...while*

```c
/* Demonstrate while and do…while loops.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int loopCount;
   int testCount;

// Statements
   loopCount = 1;
   testCount = 0;
   printf("while loop:        ");
   while (testCount++, loopCount <= 10)
      printf("%3d", loopCount++);
   printf("Loop Count:       %3d\n", loopCount);
   printf("Number of tests:  %3d\n", testCount);

   loopCount = 1;
   testCount = 0;
   printf("\ndo...while loop:  ");
   do
      printf("%3d",    loopCount++);
   while (testCount++,  loopCount <= 10);

   printf("\nLoop Count:       %3d\n", loopCount);
   printf("Number of tests:  %3d\n", testCount);
   return 0;
}  // main
```

```
Results:
while loop:          1  2  3  4  5  6  7  8  9 10
Loop Count:         11
Number of tests:    11

do...while loop:     1  2  3  4  5  6  7  8  9 10
```

continued

PROGRAM 6-8 Comparison of *while* and *do...while (continued)*

```
    Loop Count:          11
    Number of tests:     10
```

Program 6-8 Analysis Look at Statements 17 and 27 carefully. They both contain comma expressions. This technique of combining the counter and the limit test in one expression ensures that the count will be accurate. Because the value of the whole comma expression is the value of its last expression, however, the limit test must be coded last.

The results demonstrate that both loops count from one to ten. Since they are doing exactly the same job, we expect that the loop bodies would also execute the same number of times. As predicted in Table 6-1 the only difference is in the number of tests: the *while* loop control expression was evaluated 11 times; the *do...while* control expression was evaluated only 10 times.

6.6 Loop Examples

This section contains several short examples of loop applications. Each program demonstrates one or more programming concepts that you will find helpful in solving other problems.

for Loops

Example: Compound Interest

One classic loop problem is calculating the value of an investment. We want to know the value of an investment over time, given its initial value and annual interest rate. Program 6-9 displays a compound interest table.

PROGRAM 6-9 Compound Interest

```
/* Print report showing value of investment.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   double presVal;
   double futureVal;
   double rate;
   int    years;
   int    looper;
```

continued

PROGRAM 6-9 Compound Interest *(continued)*

```

// Statements
   printf("Enter value of investment:    ");
   scanf ("%lf", &presVal);
   printf("Enter rate of return (nn.n): ");
   scanf ("%lf", &rate);
   printf("Enter number of years:        ");
   scanf ("%d", &years);

   printf("\nYear    Value\n");
   printf("====    ========\n");
   for (futureVal = presVal, looper = 1;
        looper <= years;
        looper++)
       {
        futureVal = futureVal * (1 + rate/100.0);
        printf("%3d%11.2lf\n", looper, futureVal);
       } // for
   return 0;
}  // main
```

```
Results:
Enter value of investment:   10000
Enter rate of return (nn.n): 7.2
Enter number of years:       5

Year    Value
====    ========
  1     10720.00
  2     11491.84
  3     12319.25
  4     13206.24
  5     14157.09
```

Program 6-9 Analysis

This program uses a *for* loop to calculate the value of the investment at the end of each year. Each iteration adds the current year's interest to the investment and then prints its current value.

Note how we prompted the user for input, especially the decimal return rate. Things like percentage rates can be confusing to enter. Is 7.2% entered as 7.2 or .072? Make sure you give the user a sample of how the data should be entered.

Now study the way we created a caption for the reports, using equal signs to underscore the captions and spaces with width specifications to align the values in columns. This rather simple technique makes the results quite readable.

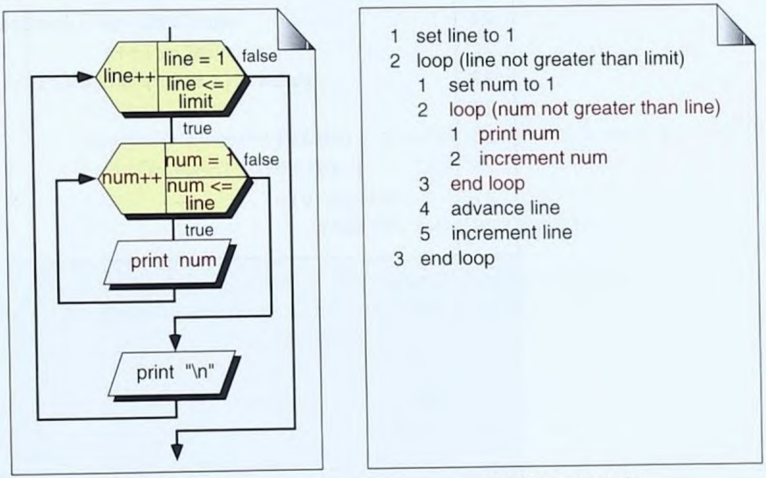
One final point. When defining multiple variables in one statement, they must all have the same type. Notice that we defined the *for* loop variables in the definition sections. To define multiple variables in a *for* statement, they must all be the same type because they are treated as a type list.

Example: Right Triangle

Let's write a program that will print a series of numbers in the form of a right triangle. We ask the user to enter a one-digit number. Each line, from the first to the limit entered by the user, then prints a number series from one to the current line number. For example, if a user enters 6, the program prints

```
1
12
123
1234
12345
123456
```

The flowchart and pseudocode for the loop are shown in Figure 6-18.



```
1 set line to 1
2 loop (line not greater than limit)
   1 set num to 1
   2 loop (num not greater than line)
      1 print num
      2 increment num
   3 end loop
   4 advance line
   5 increment line
3 end loop
```

(a) Flowchart

(b) Pseudocode

FIGURE 6-18 Print Right Triangle Flowchart and Pseudocode

The completed program is shown in Program 6-10.

PROGRAM 6-10 Print Right Triangle Using Nested *for*

```c
/* Print a number series from 1 to a user-specified limit
   in the form of a right triangle.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int limit;

// Statements
   // Read limit
   printf("\nPlease enter a number between 1 and 9: ");
   scanf("%d", &limit);

   for (int lineCtrl = 1; lineCtrl <= limit; lineCtrl++)
      {
       for (int numCtrl = 1;
                numCtrl <= lineCtrl;
                numCtrl++)
          printf("%1d", numCtrl);

       printf("\n");
      } // for lineCtrl
   return 0;
}  // main
```

```
Results:
Please enter a number between 1 and 9: 6
1
12
123
1234
12345
123456
```

Program 6-10 Analysis

Program 6-10 demonstrates the concept of a loop within a loop. Note how we use two *for* loops to print the triangle. The first or outer *for* controls how many lines we will print. The second or inner *for* writes the number series on one line. This use of nested loops is a very important programming concept.

Also note the name we used in the loops. Often programmers will use `i` and `j` to control *for* loops. We often do this ourselves. But notice how much more readable the code is when meaningful names are used. When we use `lineCtrl` rather than `i` in the outer loop, we know that the *for* is controlling the number of lines we are printing. Likewise, the name numCtrl clearly tells the reader that the *for* loop is controlling the numbers.

EXAMPLE 6-1 Print Rectangle

Now let's write a program that prints the triangle pattern in the previous example, but now filled out with asterisks to form a rectangle. For example, if a user enters 6, Program 6-11 prints the rectangle shown in the results.

PROGRAM 6-11 Print Number Series Using User-specified Limit

```
/* Print number series from 1 to a user-specified limit
   in the form of a rectangle.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int limit;

//Statements
   // Read limit
   printf("Please enter a number between 1 and 9: ");
   scanf("%d", &limit);

   for (int row = 1; row <= limit; row++)
      {
       for (int col = 1; col <= limit; col++)
            if (row >= col)
               printf("%d", col);
            else
               printf("*");
       printf("\n");
      } // for row ...
    return 0;
 } // main
```

```
Results:
Please enter a number between 1 and 9: 6
1*****
12****
123***
```

continued

PROGRAM 6-11 Print Number Series Using User-specified Limit *(continued)*

```
1234**
12345*
123456
```

Program 6-11 Analysis This program is an interesting variation of the previous program. Compare its inner loop to the inner loop in Program 6-10. This is the only part of the program that is different. The first thing you should note is that the limit test expression is different; it always goes to the maximum number of print positions. Within the inner loop, we test the column number (`col`) to determine how many digits we print on the line. If the expression is true, we print a digit. If it is false, we print an asterisk.

Example: Print Month

As a final example of a *for* loop, let's print a calendar month. In Program 6-12, the function `printMonth` receives only the start day of the month—Sunday is 0, Monday is 1 … Saturday is 6—and the number of days in the month. This is all that the program needs to print any month of the year.

PROGRAM 6-12 Print Calendar Month

```c
/* Test driver for function to print a calendar month.
      Written by:
      Date:
*/
#include <stdio.h>

// Prototype Declarations
void printMonth (int startDay, int days);

int main (void)
{
// Statements
   printMonth (2, 29);               // Day 2 is Tuesday
   return 0;
}  // main

/* ================ printMonth ================
   Print one calendar month.
      Pre   startDay is day of week relative
               to Sunday (0)
            days is number of days in month
      Post  Calendar printed
*/
void printMonth (int startDay, int days)
{
```

continued

PROGRAM 6-12 Print Calendar Month *(continued)*

```
// Local Declarations
   int weekDay;

// Statements
   // print day header
   printf("Sun Mon Tue Wed Thu Fri Sat\n");
   printf("--- --- --- --- --- --- ---\n");

   // position first day
   for (weekDay = 0; weekDay < startDay; weekDay++)
       printf("    ");

   for (int dayCount = 1; dayCount <= days; dayCount++)
       {
        if (weekDay > 6)
           {
            printf("\n");
            weekDay = 1;
           } // if
        else
            weekDay++;
        printf("%3d ", dayCount);
       } // for
   printf("\n--- --- --- --- --- --- ---\n");
   return;
}  // printMonth
```

```
Results:
Sun Mon Tue Wed Thu Fri Sat
--- --- --- --- --- --- ---
          1   2   3   4   5
  6   7   8   9  10  11  12
 13  14  15  16  17  18  19
 20  21  22  23  24  25  26
 27  28  29
--- --- --- --- --- --- ---
```

Program 6-12 Analysis This program is interesting for two reasons: first, it requires two *for* loops, one to position the printing for the first day of the month and one to print the dates. Second, the logic to control the days of the week is simple yet efficient. In an effort to eliminate one variable (weekDay), many programmers would use the modulo statement shown below to determine the day of the week.

```
(dayCount + startDay) % 7
```

Although this logic works, it is very inefficient. Using a separate variable to control the days of the week requires simple addition, which is much more efficient. It also avoids another poor programming style: using one variable to control two processes. Although using one variable to control two processes may work initially, it often leads to subtle bugs when the program is changed.

One final point. Examine the initialization in the two *for* loops. Do you see a difference? Because weekday is used in both loops, it cannot be declared local to the first loop. We must declare it in the definition section.

Never use one variable to control two processes.

while LOOPS

This section contains two examples of *while* loops. The first prints the sum of the digits in an integer entered from the keyboard. The second prints a number backwards.

Example: Print Sum of the Digits

Program 6-13 accepts an integer from the keyboard and then prints the number of digits in the integer and the sum of the digits.

PROGRAM 6-13 Print Sum of Digits

```
/* Print the number and sum of digits in an integer.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int number;
   int count = 0;
   int sum   = 0;

// Statements
   printf("Enter an integer: ");
   scanf ("%d", &number);
   printf("Your number is:   %d\n\n", number);

   while (number != 0)
      {
       count++;
       sum +=  number % 10;
       number /= 10;
```

continued

PROGRAM 6-13 Print Sum of Digits *(continued)*

```
      } // while
   printf("The number of digits is:  %3d\n", count);
   printf("The sum of the digits is: %3d\n", sum);
   return 0;
}  // main
```

```
Results:
Enter an integer: 12345
Your number is:   12345

The number of digits is:    5
The sum of the digits is:  15
```

Program 6-13 Analysis This problem requires that we "peel" off one digit at a time and add it to the total of the previous digits. We can solve this problem in a couple of ways, but by far the most straightforward is to use the modulus operator (%) to extract the rightmost digit and then to divide the number by ten to remove the right digit. For example, given the number 123, we first extract the 3 by

```
123 % 10
```

and then eliminate it by dividing by 10 to give 12. Note that since both the dividend and the divisor are integers, the result is an integer. We loop until there is only one digit left, at which time any digit divided by ten will result in zero and the loop terminates.

Example: Print Number Backward

Now let's look at an example of a loop that prints a number backward. We can solve this problem in several ways. Perhaps the easiest is to simply use modulo division, as shown in Program 6-14.

PROGRAM 6-14 Print Number Backward

```
/* Use a loop to print a number backward.
     Written by:
     Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   long num;
   int  digit;

```

continued

PROGRAM 6-14 Print Number Backward *(continued)*

```
// Statements
   printf("Enter a number and I'll print it backward: ");
   scanf ("%d", &num);

   while (num > 0)
      {
       digit = num % 10;
       printf("%d", digit);
       num   = num / 10;
      } // while
   printf("\nHave a good day.\n");
   return 0;
}  // main
```

```
Results:
Enter a number and I'll print it backward: 12345678
87654321
Have a good day.
```

do...while LOOPS

This sections contains an example of a program that uses a *do...while* loop. While it is a simple problem, it demonstrates a classic data-validation technique.

EXAMPLE 6-2 Data Validation

Now we'll write a program that reads an integer consisting of only zeros and ones (a binary number) and converts it to its decimal equivalent. Provide a function that ensures that the number entered is a binary number. The design for this program is shown in Figure 6-19.

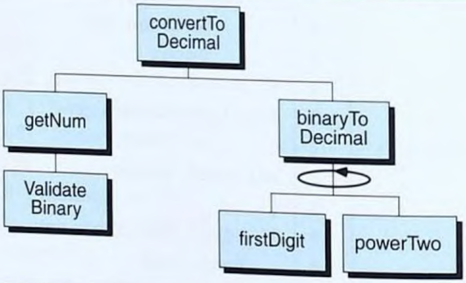


FIGURE 6-19 Design for Binary to Decimal

This design follows a classic program design: input–process–output. The input function is `getNum`; the process function is `binaryToDecimal`; and the output function is `printResults`. Study the design of `binaryToDecimal`. Note that it uses a loop and calls `firstDigit` and `powerTwo` in turn within the loop. How would you implement this loop? Think about it for a minute, and then look at Statement 65 in Program 6-15 to see how we did it. If you are not sure of your binary arithmetic and binary conversions, study Appendix D, "Numbering Systems."

PROGRAM 6-15 Convert Binary to Decimal

```
 1  /* Convert a binary number to a decimal number.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6  #include <stdbool.h>
 7
 8  // Function Declarations
 9  long long getNum          (void);
10  long long binaryToDecimal (long long binary);
11  long long powerTwo        (long long num);
12  long long firstDigit      (long long num);
13  bool      validateBinary  (long long binary);
14
15  int main (void)
16  {
17  // Local Declarations
18     long long binary;
19     long long decimal;
20
21  // Statements
22     binary   = getNum ();
23     decimal  = binaryToDecimal (binary);
24     printf("The binary number was: %lld", binary);
25     printf("\nThe decimal number is: %lld", decimal);
26     return 0;
27  }  // main
28
29  /* ==================== getNum ====================
30     This function reads and validates a binary number
31     from the keyboard.
32        Pre   nothing
33        Post  a valid binary number is returned
34  */
```

continued

PROGRAM 6-15 Convert Binary to Decimal *(continued)*

```
long long getNum (void)
{
// Local Declarations
   bool      isValid;
   long long binary;

// Statements
   do
    {
     printf("Enter a binary number (zeros and ones): ");
     scanf ("%lld", &binary);
     isValid = validateBinary (binary);
     if (!isValid)
        printf("\a\aNot binary. Zeros/ones only.\n\n");
    } while (!isValid);
   return binary;
} // getNum

/* ================ binaryToDecimal ==================
   Change a binary number to a decimal number.
     Pre  binary is a number containing only 0 or 1
     Post returns decimal number
*/
long long binaryToDecimal (long long binary)
{
// Local Declarations
   long long  decimal;

// Statements
   decimal = 0;
   for (int i = 0; binary != 0; i++)
       {
        decimal += firstDigit (binary) * powerTwo (i);
        binary /= 10;
       } // for i
   return decimal;
} // binaryToDecimal

/* ================ validateBinary ==================
   Check the digits in a binary number for only 0 and 1.
      Pre   binary is a number to be validated
      Post  returns true if valid; false if not
*/
```

continued

PROGRAM 6-15 Convert Binary to Decimal *(continued)*

```
bool validateBinary (long long binary)
{
// Statements
  while (binary != 0)
    {
     if (!(binary % 10 ==  0 || binary % 10 ==  1))
        return false;
     binary /= 10;
    } // while
  return true;
} // validateBinary

/* ==================== powerTwo ====================
  This function raises 2 to the power num
    Pre   num is exponent
    Post  Returns 2 to the power of num
*/
long long powerTwo (long long num)
{
// Local Declarations
  long long  power = 1;

// Statements
  for (int i = 1; i <= num; i++)
        power *= 2;
  return power;
}  // powerTwo

/* ================== firstDigit ==================
  This function returns the right most digit of num
    Pre   the integer num
    Post  the right digit of num returned
*/
long long firstDigit (long long num)
{
// Statements
  return (num % 10);
} // firstDigit
```

```
Results:
Enter a binary number (zeros and ones): 10001
The binary number was: 10001
The decimal number is: 17
```

Program 6-15 Analysis This program has several interesting aspects. First, note the data validation that we use to ensure that the binary number that we read consists of nothing but zeros and ones. We used this series of modulus and divide statements previously in Program 6-13 and Program 6-14.

Then, note how we enclosed the call to the validation function in a *do...while* that allows us to keep reading input until the user gives us a binary number. Again, note how we display an error message when the number is not valid. This is a standard data validation technique.

Next, study the `binaryToDecimal` function that converts the binary number to its decimal value. Note that when we extract a digit, it is either a zero or a one. We then multiply the extracted digit by 2 raised to the digit position we are currently evaluating, which gives us the binary value of that digit's position in the binary number. The value is then added to the decimal number. Of course, if the digit is a zero, then the product is zero and the value is unchanged. It is only when the digit is a one that we add to the decimal number.

Finally, note that throughout this program we used *long long* for the binary and decimal number. We did this because the decimal representation of a binary number can get very big very fast. On a personal computer, *long* might not be able to hold a large binary number.

6.7 Other Statements Related to Looping

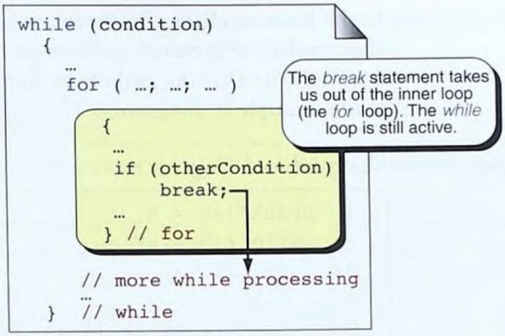
Three other C statements are related to loops: *break*, *continue*, and *goto*. The last statements, the *goto*, is not valid for structured programs and therefore is not discussed in this text.

break

The first statement is ***break***. We previously saw the *break* statement when we discussed ***switch*** in Chapter 5. In a loop, the *break* statement causes a loop to terminate. It is the same as setting the loop's limit test to false. If we are in a series of nested loops, *break* terminates only the inner loop—the one we are currently in. Figure 6-20 shows how *break* transfers control out of an inner *for* loop and continues with the next statement in the *while*. Note that *break* statement needs a semicolon.

The *break* statement can be used in any of the loop statements—*while*, *for*, and *do...while*—and in the selection *switch* statement. However, good structured programming limits its use to the *switch* statement. It should not be used in any of the looping statements. If you feel you must use *break*, reexamine your design. You will usually find that you have not properly structured your logic.[1]

[1] Although this statement is generally true, as you study advanced programming concepts, such as parsing, you will find that *break* and *continue* are used to simplify the code.



```
while (condition)
  {
   …
   for ( …; …; … )
     {
      …
      if (otherCondition)
         break;
      …
     } // for

   // more while processing
   …
  } // while
```

The *break* statement takes us out of the inner loop (the *for* loop). The *while* loop is still active.

FIGURE 6-20 *break* and Inner Loops

Program 6-16 shows two examples of poor loop situations and how to restructure them so that the *break* is not needed. The *for* statement shows a never-ending loop. As coded, there is no way to terminate the loop without the *break*. Although the *break* works, better documentation and style puts the limiting condition as the second expression in the *for* statement. After all, that is the use of the limit expression in the first place.

PROGRAM 6-16 The *for* and *while* as Perpetual Loops

```
// A bad loop style
for ( ; ; )
  {
   …
   if (condition)
      break;
  } // for
```

```
// A better loop style
for ( ; !condition ; )
  {
   …
  } // for
```

```
while (x)
  {
   …
   if (condition)
      break;
   else
      …
  } // while
```

```
while  (x && !condition)
  {
   …
   if (!condition)
      …;
  } // while
```

Even if the *break* condition is in the middle of a block, it can be removed to the limit condition, as shown in the *while* example in Program 6-16. Note that in this case, we complement the condition, which then executes the *else* logic when the limit condition has not been reached.

Sometimes the limit condition is so complex that it cannot easily be put in the limit condition. In these cases, a flag is used. A flag is a logical variable that tracks the presence or absence of an event. In this case, the flag is set to 1 to indicate that the end of the loop has been reached. This is shown in the *while* example in Program 6-17.

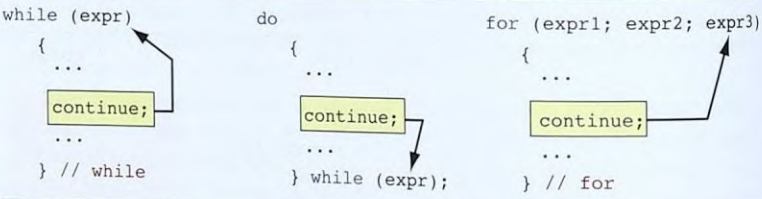
PROGRAM 6-17 Using a *break* Flag

```
breakFlag = 0;
while (!breakFlag)
   {
    …
    if (x && !y || z)                 // Complex limit test
       breakFlag = 1;
    else
       …;
   } // while
```

On the other hand, too many flags can make a function overly complex. The use of *break* and flags needs to be tempered with simplicity and clarity of logic. Finally, note that the flag is called `breakFlag`, which is a generic name. You should choose a more descriptive name, such as `accountFlag` or `timeLimitFlag`.

continue

The ***continue*** statement does not terminate the loop but simply transfers to the testing expression in *while* and *do…while* statements and transfers to the updating expression in a *for* statement. These jumps are shown in Figure 6-21. Although the transfer is to different positions in pretest and post-test loops, both can be logically thought of as a jump to the end of the loop's body.



```
while (expr)
  {
   ...
   continue;
   ...
  } // while
```

```
do
  {
   ...
   continue;
   ...
  } while (expr);
```

```
for (expr1; expr2; expr3)
  {
   ...
   continue;
   ...
  } // for
```

FIGURE 6-21 The *continue* Statement

The use of the *continue* statement is also considered unstructured programming. If you think that you need a *continue*, your algorithm may not be well structured. A little study will show how to eliminate it. Program 6-18 contains a common *continue* example found in many textbooks. In this function, the assignment is to read data and return the average of nonzero numbers read. In other words, it skips zeros. Note how simply reversing the conditional test eliminates the need for the *continue*.

PROGRAM 6-18 *continue* Example

```
float readAverage (void)
{
// Local Declarations
int   count = 0;

int   n;
float sum = 0;

// Statements
while(scanf("%d",&n)
        != EOF)
    {
     if (n == 0)
         continue;
     sum += n;
     count++;
    }  // while

 return (sum / count);
} // readAverage
```

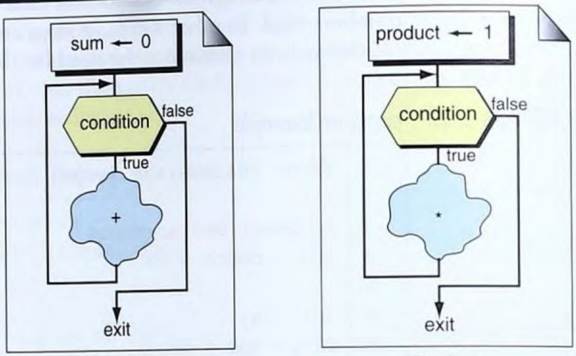
```
float readAverage (void)
{
// Local Declarations
int   count = 0;

int   n;
float sum = 0;

// Statements
while(scanf("%d",&n)
        != EOF)
    {
     if (n != 0)
        {
         sum += n;
         count++;
        } // if
    } // while
 return (sum / count);
} // readAverage
```

6.8 Looping Applications

In this section, we examine four common applications for loops: summation, product, smallest or largest, and inquiries. Although the uses for loops are virtually endless, these problems illustrate many common applications. Note that a common design runs through all looping applications. With few exceptions, each loop contains initialization code, looping code, and disposition code. Disposition code handles the result of the loop, often by printing it, other times by simply returning it to the calling function.

Summation

As you have seen, we can add two or three numbers very easily. But how can we add many numbers or a variable series of numbers? The solution is simple: use the add operator in a loop. The concept of **summation** is graphically shown in Figure 6-22.



Summation

Product

FIGURE 6-22 Summation and Product Loops

A sum function has three logical parts: (1) initialization of any necessary working variables, such as the sum accumulator; (2) the loop, which includes the summation code and any data validation code (for example, "only nonzero numbers are to be considered"); and (3) the disposition code to print or return the result. Program 6-19 is a loop function that reads a series of numbers from the keyboard and returns their sum. In each loop, we read the next number and add it to the accumulator, sum. A similar application, counting, is a special case of summation in which we add 1 to a counter instead of adding the number we read to an accumulator.

PROGRAM 6-19 Sum to EOF Function

```
/* Read a series of numbers, terminated by EOF, and
   return their sum to the calling program.
      Pre   nothing
      Post  data read and sum returned
*/
int sumEOF (void)
{
// Local Declarations
   int nmbr;
   int sum;

// Statements
   sum = 0;
   printf ("Please enter an integer:    ");

   while (scanf("%d", &nmbr) != EOF)
```

continued

PROGRAM 6-19 Sum to EOF Function *(continued)*

```
       {
        sum += nmbr;
        printf("Next integer <EOF> to stop: ");
       } // while
    return sum;
}  // sumEOF
```

Powers

Just as we can add a series of numbers in a loop, we can perform any mathematical operation. A product loop is useful for two common applications: raising a number to a power and calculating the factorial of a number. For example, Program 6-20 shows a function to return x^n. Notice that this function also includes initialization logic to validate the parameter list. If either of the parameters is invalid, we return zero as an error indicator.

PROGRAM 6-20 Powers Function

```
/* Raise base to an integral power, exp. If the
   exponent is zero, return 1.
      Pre   base & exp are both positive integer values
      Post  return either (a) the result of raising the
                base to the exp power
            or (b) zero if the parameters are invalid
*/
int powers (int base, int exp)
{
// Local Declarations
   int result = 1;

// Statements
   if (base < 1 || exp < 0)
      // Error Condition
      result = 0;
   else
      for  (int i = 1; i <= exp; i++)
           result *= base;
   return result;
}  // powers
```

The summation and powers examples demonstrate a subtle point: initialization must be based on the application; it is not the same for every problem. For summation, the initialization sets the accumulator to 0. For product-based applications, such as powers, we must initialize it to 1. If we use 0, then the result is 0 because 0 multiplied by anything is 0.

To find the sum of a series, the result is initialized to 0; to find the product of a series, the result is initialized to 1.

Smallest and Largest

We often encounter situations in which we must determine the smallest or largest among a series of data. This is also a natural looping structure.

We can write a statement to find the smaller of two numbers. For example,

```
result = a  < b ? a : b;
```

But how can we find the smallest of several numbers? We simply put the same statement inside a loop. Each iteration then tests the current smallest to the next number. If this new number is smaller than the current smallest, we replace the smallest. In other words, we determine the smallest number by looping through a series while remembering the smallest number the loop has found. This concept is shown in Figure 6-23.

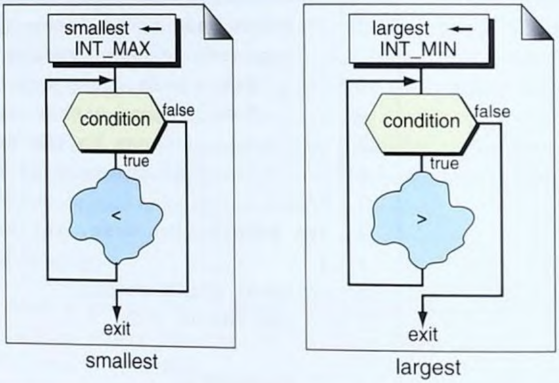


FIGURE 6-23 Smallest and Largest Loops

To find the smallest of a series, the initialization sets the initial value of `smallest` to the largest possible value for its type. In a C program, the largest value for an integer is given as `INT_MAX`, which is found in the limits library (*limits.h*). The loop then proceeds to read a series of numbers and tests each one against the previously stored `smallest` number. Since `smallest` starts with the maximum integer value, the first read number automatically becomes `smallest`. Thereafter, the result depends entirely on the data being read. The disposition simply returns the `smallest` value found.

Program 6-21 is the C implementation of `smallest`.

PROGRAM 6-21 Smallest to EOF Function

```
/* Read a series of numbers, terminated by EOF, and
   pass the smallest to the calling program.
      Pre   nothing
      Post  data read and smallest returned
*/
int smallestEOF (void)
{
// Local Declarations
   int numIn;
   int smallest;

// Statements
   smallest = INT_MAX;         // requires <limits.h>

   printf("Please enter an integer: ");

   while (scanf("%d", &numIn) != EOF)
      {
       if (numIn < smallest)
           smallest = numIn;
       printf("Enter next integer <EOF> to stop: ");
      } // while
    return smallest;
}   // smallestEOF
```

We can find the largest number by simply reversing the less than operator in the expression, making it greater than. We would also need to set the variable, renamed `largest`, to `INT_MIN`.

To find the largest, we need to initialize the result (a variable named **`largest`**) variable to a very small number, such as **`INT_MIN`**.

To find the smallest, we need to initialize the result (a variable named **`smallest`**) to a very large number, such as **`INT_MAX`**.

Inquiries

An **inquiry** is simply a question asked of the computer program. In programming, we often encounter one of two basic inquiry types: **any** and **all**. We use the inquiry type *any* when we have a list of data and we want to know if *at least one* of them meet a given criteria. The answer to the inquiry is yes if one or more data meet the criteria. The answer is no if none of the data meet the criteria.

We use *all* when we have a list of data and we want to make sure that *each and every one* of them meet some specified criteria. The concept of *any* and *all* in an inquiry is shown in Figure 6-24.

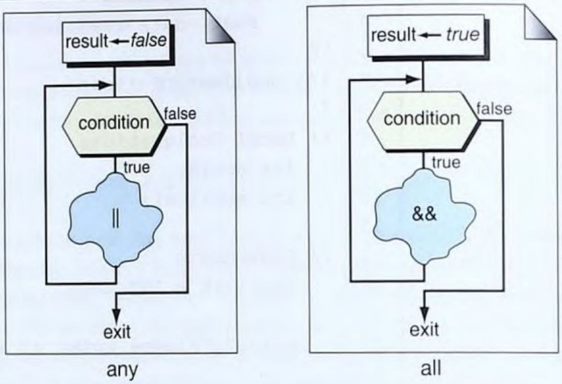


FIGURE 6-24 *any* and *all* Inquiries

The any Inquiry

To determine if *one* item in a series satisfies a requirement, we can test them using *or* as shown in the next example.

```
result = (a == condition) || (b == condition);
```

If either or both of the equal expressions are true, result is set to *true*. This works fine for testing the first two items. To test the third item in the series, however, we need to include the result of the first two. This is easily done with the next example.

```
result = result || (c == condition);
```

Study this test carefully. If the previous tests were all *false*, then result contains *false* when the statement is executed. If the equal expression is *false*, then result is assigned *false*; however, if the equal expression is *true*, then result is assigned *true*. By placing this expression in a loop, we can test all of the values in the series. For example, to test a series to determine if any of the values are positive, we can use the following code.

```
result = false;
while (scanf("%d", &number)
   {
    result = result || number > 0;
   } // while
```

This code produces an accurate answer to the inquiry. However, it can be very inefficient especially if the series is long. For example, suppose that the first item tested is greater than 0. In this case, it is not necessary to test the rest of the list; we already know the answer.

Program 6-22 is an example of an *any* inquiry. It reads a series of numbers and checks to see if any of the numbers in the list are greater than zero. The function terminates and returns *true* as soon as a positive number is read.

PROGRAM 6-22 anyPositive to EOF Function

```
/* Read number series to determine if any positive.
      Pre  nothing
      Post return true if any number > zero
           return false if all numbers <= zero
*/
bool anyPositiveEOF (void)
{
// Local Declarations
   bool anyPositive = false;
   int  numIn;

// Statements
   printf("Determine if any number are positive\n");
   printf("Enter first number: ");
   while (scanf("%d", &numIn) != EOF)
      {
       anyPositive = numIn > 0;
       if (anyPositive)
           return true;
       printf("Enter next number:  ");
      } // while
    return false;
 } // anyPositiveEOF
```

Program 6-22 Analysis Note that `anyPositive` is a logical variable (*bool*) and is initialized to *false*. Because we do not need to read the entire number series, we can use a simple test to set `anyPositive`. If a positive number was entered, we return *true*. If not, we prompt for the next number and loop back to the *while* expression.

To use the *bool* type, we must include the *stdbool.h* library in the program. This library uses a define macro to declare the *bool* type and equate it to the C Boolean type, *_bool*. It also defines *true* and *false*.

The `all` Inquiry

To determine if two items in a series satisfy a requirement, we can test them using *and* as shown in the next example.

```
result = (a == condition) && (b == condition);
```

If both of the equal expressions are true, result is set to *true*; if either of them is *false*, the result is *false*. This works fine for testing the first two items. To test the third item in the series, however, we need to include the result of the first two. This is easily done with the next example.

```
result = result && (c == condition);
```

Study this test carefully. If the previous tests were all *true*, then result contains *true* when the statement is executed. If the equal expression is also *true*, then result is assigned *true*; however, if the equal expression is *false*, then result is assigned *false*. By placing this expression in a loop, we can test all of the values in the series. For example, to test a series to determine if all of the values are positive, we can use the following code.

```
result = true;
while (scanf("%d", &number)
   {
    result = result && number > 0;
   } // while
```

Once again, for efficient processing we may not need to examine all of the numbers in the series. As soon as a nonpositive number is found, we know the result and can return *false*.

This function is seen in Program 6-23.

PROGRAM 6-23 All Positive Function

```
/* Read number series, and determine if all are positive.
      Pre   nothing
      Post  return true if all numbers > zero
            return false if any numbers <= zero
*/
bool allPositiveEOF (void)
{
// Local Declarations
   bool allPositive = true;
   int  numIn;

// Statements
   printf("Determine if all numbers are positive\n");
   printf("Enter first number: ");
   while (allPositive && (scanf("%d", &numIn) != EOF))
      {
```

continued

PROGRAM 6-23 All Positive Function *(continued)*

```
       allPositive =  numIn > 0;
       if (!allPositive)
           return false;
       printf("Enter next number:  ");
      } // while
   return true;
} // allPositiveEOF
```

Once again you should study how we initialize the Boolean flags in these two programs. To test for any item, we start with the flag *false*. If and when it becomes *true*, the result has been determined. Similarly, to test for all items, we initialize the Boolean flag to *true*. If and when it becomes *false*, the result has been determined.

To answer an *any* inquiry, the result is initialized to *false*; to answer an *all* inquiry, the result is initialized to *true*.

6.9 Recursion

In general, programmers use two approaches to writing repetitive algorithms. One approach uses loops; the other uses recursion. **Recursion** is a repetitive process in which a function calls itself. Some older languages do not support recursion. One major language that does not is COBOL.

Iterative Definition

To study a simple example, consider the calculation of a factorial. The factorial of a number is the product of the integral values from 1 to the number. This definition is shown in Formula 6-1.

$$\text{factorial}\ (n) = \begin{bmatrix} 1 & \text{if } n = 0 \\ n * (n-1) * (n-2) \ldots 3 * 2 * 1 & \text{if } n > 0 \end{bmatrix}$$

FORMULA 6-1 Iterative Factorial Definition

This definition is iterative. A repetitive function is defined iteratively whenever the definition involves only the parameter(s) and not the function itself. We can calculate the value of factorial (4) using Formula 6-1 as follows:

```
factorial (4) = 4 * 3 * 2 * 1 = 24
```

Recursive Definition

A repetitive function is defined recursively whenever the function appears within the definition itself. For example, the factorial function can be defined recursively, as shown in Formula 6-2.

$$\text{factorial}\ (n) = \begin{bmatrix} 1 & \text{if } n = 0 \\ n * \text{factorial}\ (n - 1) & \text{if } n > 0 \end{bmatrix}$$

FORMULA 6-2 Recursive Factorial Definition

The decomposition of factorial (3), using Formula 2, is shown in Figure 6-25. Study this figure carefully, and note that the recursive solution for a problem involves a two-way journey: first we decompose the problem from the top to the bottom, and then we solve it from the bottom to the top.

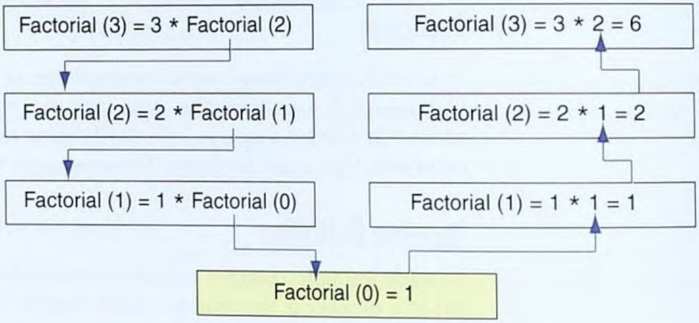

Factorial (3) = 3 * Factorial (2)
Factorial (2) = 2 * Factorial (1)
Factorial (1) = 1 * Factorial (0)
Factorial (0) = 1
Factorial (1) = 1 * 1 = 1
Factorial (2) = 2 * 1 = 2
Factorial (3) = 3 * 2 = 6

FIGURE 6-25 Factorial (3) Recursively

Judging by the above example, the recursive calculation looks much longer and more difficult. So why would we want to use the recursive method? We use it because, although the recursive calculation looks more difficult when using paper and pencil, it is often a much easier and more elegant solution when we use computers. Additionally, it offers a conceptual simplicity to the creator and the reader.

Iterative Solution

Let's write a function to solve the factorial problem iteratively. This solution usually involves using a loop, as shown in Program 6-24.

PROGRAM 6-24 Iterative Factorial Function

```
/* Calculate the factorial of a number using a loop.
   There is no test that the result fits in a long.
      Pre   n is the number to be raised factorially
      Post  result is returned
*/
long factorial (int n)
{
// Local Declarations
   long factN = 1;

// Statements
   for (int i = 1; i <= n; i++)
       factN = factN * i;
   return factN;
} // factorial
```

Recursive Solution

The recursive solution to factorial is shown in Program 6-25. This program does not need a loop; the concept itself involves repetition.

PROGRAM 6-25 Recursive Factorial

```
 1  /* Calculate factorial of a number using recursion.
 2     There is no test that the result fits in a long.
 3        Pre   n is the number being raised factorially
 4        Post  result is returned
 5  */
 6  long factorial (int n)
 7  {
 8  // Statements
 9     if (n == 0)
10        return 1;
11     else
12        return (n * factorial (n - 1));
13  } // factorial
```

In the recursive version, we let the function `factorial` call itself, each time with a different set of parameters. Figure 6-26 shows the recursive execution with the parameters for each individual call.

Designing Recursive Functions

All recursive functions have two elements: each call either solves *one* part of the problem or it *reduces* the size of the problem. In Program 6-25, statement 10 solves a small piece of the problem—`factorial (0)` is 1.

Statement 12, on the other hand, reduces the size of the problem by recursively calling the factorial with $n - 1$. Once the solution to `factorial (n - 1)` is known, Statement 12 provides part of the solution to the general problem by returning a value to the calling function.

As we see in Statement 12, the general part of the solution is the recursive call: Statement 12 calls its own function to solve the problem. We also see this in Figure 6-26. At each recursive call, the size of the problem is reduced from the factorial of 3, to 2, 1, and finally to factorial 0.

The statement that "solves" the problem is known as the **base case**. *Every recursive function must have a base case*. The rest of the function is known as the **general case**. In our factorial example, the base case is `factorial (0)`; the general case is `n * factorial (n - 1)`. The general case contains the logic needed to reduce the size of the problem.

Every recursive call must either solve part of the problem or reduce the size of the problem.

In the factorial problem, once the base case has been reached, the solution begins. The program has found one part of the answer and can return that part to the next more general statement. Thus, in Program 6-25, after the program has calculated that `factorial (0)` is 1, then it returns value 1. That leads to solving the next general case,

```
factorial (1) → 1 * factorial (0) → 1 * 1 → 1
```

The program now returns the value of `factorial (1)` to the more general case, `factorial (2)`, which we know to be

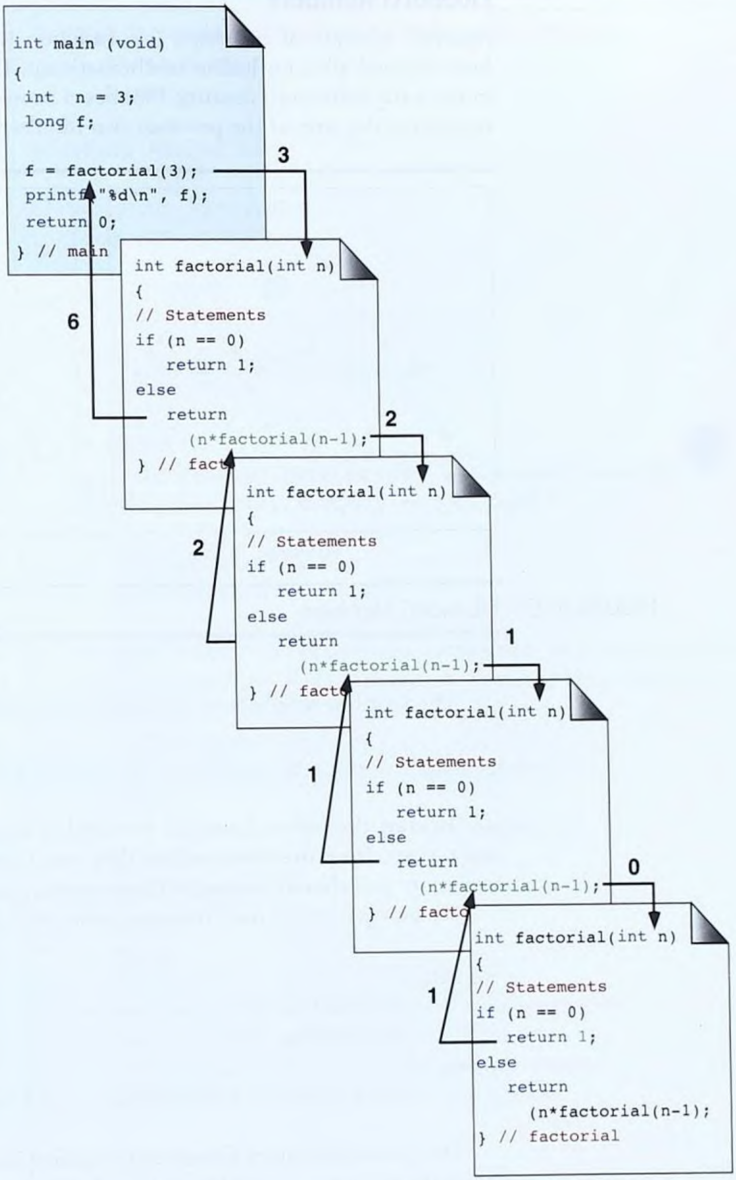
```
factorial (2) → 2 * factorial (1) → 2 * 1 → 2
```

As the program solves each general case in turn, the program can solve the next higher general case, until it finally solves the most general case, the original problem.

The following are rules for designing a recursive function:

1. First, determine the base case.
2. Then, determine the general case.
3. Finally, combine the base case and general case into a function.

In combining the base and general cases into a function, we must pay careful attention to the logic. Each call must reduce the size of the problem and move it toward the base case. The base case, when reached, must terminate without a call to the recursive function; that is, it must execute a *return*.



```
int main (void)
{
  int  n = 3;
  long f;

  f = factorial(3);
  printf("%d\n", f);
  return 0;
} // main
```

```
int factorial(int n)
{
// Statements
if (n == 0)
   return 1;
else
   return
      (n*factorial(n-1));
} // factorial
```

```
int factorial(int n)
{
// Statements
if (n == 0)
   return 1;
else
   return
      (n*factorial(n-1));
} // factorial
```

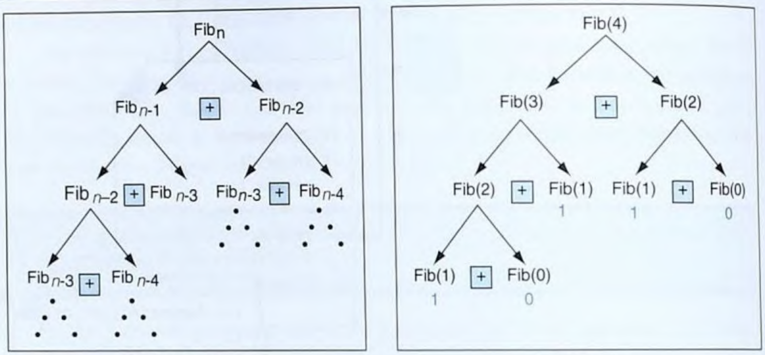
```
int factorial(int n)
{
// Statements
if (n == 0)
   return 1;
else
   return
      (n*factorial(n-1));
} // factorial
```

```
int factorial(int n)
{
// Statements
if (n == 0)
   return 1;
else
   return
      (n*factorial(n-1));
} // factorial
```

FIGURE 6-26 Calling a Recursive Function

Fibonacci Numbers

Another example of recursion is a function that generates Fibonacci numbers. Named after an Italian mathematician, Leonardo Fibonacci, who lived in the early thirteenth century, Fibonacci numbers are a series in which each number is the sum of the previous two numbers (Figure 6-27).



(a) Fib(n)

(b) Fib(4)

FIGURE 6-27 Fibonacci Numbers

The first few numbers in the Fibonacci series are

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

To start the series, however, we need to know the first two numbers. As you can see from the above series, they are 0 and 1. Since we are discussing recursion, you should recognize these two numbers as the base cases.

We can generalize the Fibonacci series as follows:

```
Given:
        $Fibonacci_0 = 0$
        $Fibonacci_1 = 1$
Then
        $Fibonacci_n = Fibonacci_{n-1} + Fibonacci_{n-2}$
```

The generalization of Fibonacci$_4$ is shown in Figure 6-27. Figure 6-27 (a) shows the components of Fibonacci$_4$ using a general notation. Figure 6-27 (b) shows the components as they would be called to generate the numbers in the series.

To determine Fibonacci$_4$, we can start at 0 and move up until we have the number, or we can start at Fibonacci$_4$ and move down to zero. The first technique is used in the iterative solution; the second is used in the recursive solution, which is shown in Program 6-26.

PROGRAM 6-26 Recursive Fibonacci

```
/* This program prints out a Fibonacci series.
      Written by:
      Date:
*/
#include <stdio.h>

// Function Declaration
   long fib (long num);

int main (void)
{
// Local Declarations
   int seriesSize;

// Statements
   printf("This program prints a Fibonacci series.\n");
   printf("How many numbers do you want? ");
   scanf ("%d", &seriesSize);
   if (seriesSize < 2)
      seriesSize = 2;

   printf("First %d Fibonacci numbers: \n", seriesSize);
   for (int looper = 0; looper < seriesSize; looper++)
      {
       if (looper % 5)
           printf(", %8ld", fib(looper));
       else
           printf("\n%8ld", fib(looper));
      }
   printf("\n");
   return 0;
}  // main

/* ================ fib ==================
   Calculates the nth Fibonacci number.
      Pre   num identifies Fibonacci number
      Post  returns nth Fibonacci number
*/
long fib (long num)
{
// Statements
   if (num == 0 || num == 1)
      // Base Case
      return num;
```

continued

PROGRAM 6-26 Recursive Fibonacci *(continued)*

```

   return (fib (num - 1) + fib (num - 2));
}  // fib
```

```
Results:
This program prints a Fibonacci series.
How many numbers do you want? 30
First 30 Fibonacci numbers:
         0,        1,        1,        2,         3
         5,        8,       13,       21,        34
        55,       89,      144,      233,       377
       610,      987,     1597,     2584,      4181
      6765,    10946,    17711,    28657,     46368
     75025,   121393,   196418,   317811,    514229
```

Program 6-26 Analysis Compare `fib` in Program 6-26 with the solution in Figure 6-27. To determine the fourth number in the series, we call `fib` with `num` set to 4. To determine the answer requires that `fib` be called recursively eight times, as shown in Figure 6-27, which with the original call gives us a total of nine calls.

This sounds reasonable. Now, how many calls does it take to determine Fibonacci$_5$? The answer is 15 (Table 6-2). As you can see from Table 6-2, the number of calls goes up quickly as we increase the size of the Fibonacci number we are calculating.

No.	Calls	No.	Calls
1	1	11	287
2	3	12	465
3	5	13	753
4	9	14	1,219
5	15	15	1,973
6	25	20	21,891
7	41	25	242,785
8	67	30	2,692,573
9	109	35	29,860,703
10	177	40	331,160,281

TABLE 6-2 Fibonacci Run Time

Table 6-2 leads us to the conclusion that a recursive solution to calculate Fibonacci numbers is not realistic.

Limitations of Recursion

We have introduced only the briefest explanation of recursion in this section. We have not attempted to demonstrate *how* recursion works. To understand how it works, you will need to study data structures and concepts that are beyond the scope of this text.

On the other hand, you do need to understand the two major limitations of recursion. First, recursive solutions may involve extensive overhead because they use function calls. Second, each time you make a call, you use up some of your memory allocation. If the recursion is deep—that is, if the program has a large number of recursive calls—then you may run out of memory. Both the factorial and Fibonacci numbers solutions are better developed iteratively.

Does this mean that iterative solutions are always better than recursive functions? The answer is definitely no. Many algorithms are easier to implement recursively and are efficient. When you study data structures, you will study many of them. Unfortunately, most of those algorithms require data structures beyond the scope of this text.

The Towers Of Hanoi

One classic recursion problem, the Towers of Hanoi, is relatively easy to follow, is efficient, and uses no complex data structures. Let's look at it.

According to legend, the monks in a remote mountain monastery knew how to predict when the world would end. They had a set of three diamond needles. Stacked on the first diamond needle were 64 gold disks of decreasing size. The monks moved one disk to another needle each hour, subject to the following rules:

1. Only one disk could be moved at a time.
2. A larger disk must never be stacked above a smaller one.
3. One and only one auxiliary needle could be used for the intermediate storage of disks.

The legend said that when all 64 disks had been transferred to the destination needle, the stars would be extinguished and the world would end. Today we know that we need to have $2^{64} - 1$ moves to do this task. Figure 6-28 shows the Towers of Hanoi with only three disks.

This problem is interesting for two reasons. First, the recursive solution is much easier to code than the iterative solution would be. This is often the case with good recursive solutions. Second, its solution pattern differs from the simple examples we have been discussing. As you study the Towers solution, note that after each base case, we return to a decomposition of the general case for several steps. In other words, the problem is divided into several subproblems, each of which has a base case, moving one disk.

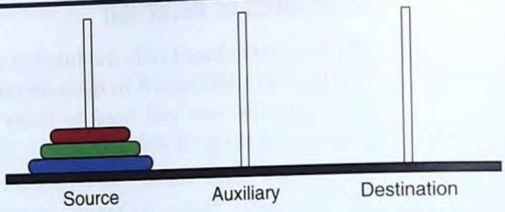


FIGURE 6-28 Towers of Hanoi—Start Position

Recursive Solution For Towers Of Hanoi

To solve this problem, we must study the moves to see if we can find a pattern. We will use only three disks because we do not want the world to end! First, imagine that we have only one disk to move. This very simple case involves only one step as shown in Case 1.

Case 1:	Move one disk from source to destination needle.

Now imagine that we have to move two disks. First, the top disk is moved to the auxiliary needle. Then the second disk is moved to the destination. Finally, the first disk is moved to the top of the second disk on the destination. These three steps are shown in Case 2.

Case 2:	Move one disk to auxiliary needle. Move one disk to destination needle. Move one disk from auxiliary to destination needle.

Figure 6-29 traces the steps for two disks.

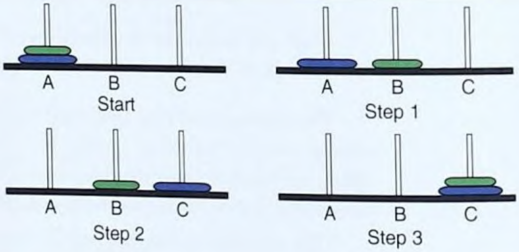


FIGURE 6-29 Towers Solution for Two Disks

We are now ready to study the case for three disks. Its solution is seen in Figure 6-30.

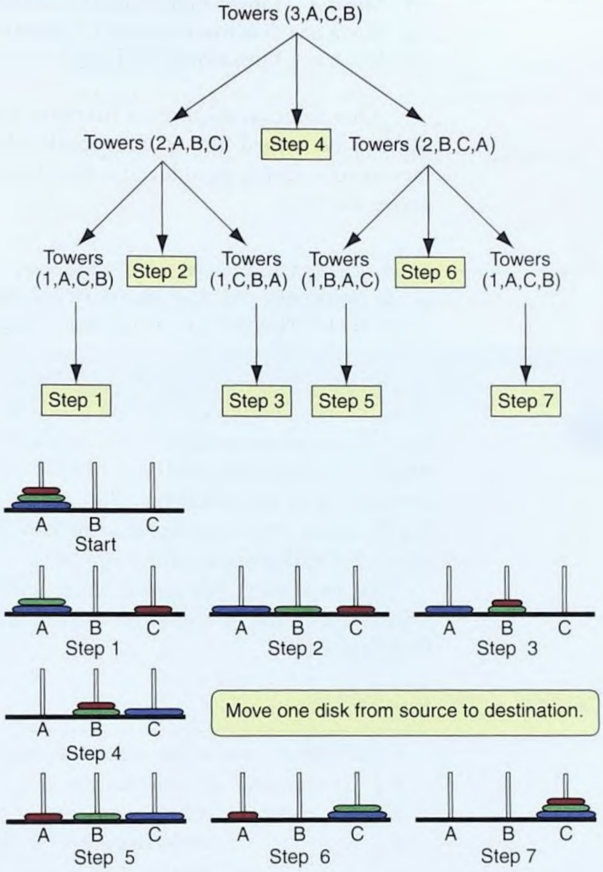


FIGURE 6-30 Towers of Hanoi Solution for Three Disks

The first three steps move the top two disks from the source to the auxiliary needle. (To see how to do this, refer to Case 2.) In Step 4, we move the bottom disk to the destination. We now have one disk in place. This is an example of Case 1. It then takes three more steps to move the two disks on the auxiliary needle to the destination. These steps are shown in Case 3.

Case 3: Move two disks from source to auxiliary needle.
Move one disk from source to destination needle.
Move two disks from auxiliary to destination needle.

We are now ready to generalize the problem.

1. Move $n - 1$ disks from source to auxiliary needle. **General Case**
2. Move one disk from source to destination needle. **Base Case**
3. Move $n - 1$ disks from auxiliary to destination needle. **General Case**

Our solution requires a function with four parameters: the number of disks to be moved, the source needle, the destination needle, and the auxiliary needle. Using pseudocode, the three moves in the generalization shown above are then

```
1 Call Towers (n - 1, source, auxiliary, destination)
2 Move one disk from source to destination
3 Call Towers (n - 1, auxiliary, destination, source)
```

Study the third step carefully. After we complete the move of the first disk, the remaining disks are on the auxiliary needle. We need to move them from the auxiliary needle to the destination. In this case, the original source needle becomes the auxiliary needle. Remember that the positions of the parameters in the *called* function are source, destination, auxiliary. The calling function must remember which of the three needles is the source and which is the destination for each call.

We can now put these three calls together with the appropriate print statements to show the moves. The complete function is shown in Program 6-27.

PROGRAM 6-27 Towers of Hanoi

```
/* Move one disk from source to destination through
   the use of recursion.
      Pre   The tower consists of n disks--source,
            destination, & auxiliary towers given
      Post  Steps for moves printed
*/
void towers (int  n,    char source,
                  char dest, char auxiliary)
{
// Local Declarations
   static int step = 0;

// Statements
   printf("Towers (%d, %c, %c, %c)\n",
                     n, source, dest, auxiliary);
   if (n == 1)
       printf("\t\t\t\tStep %3d: Move from %c to %c\n",
               ++step, source, dest);
   else
```

continued

PROGRAM 6-27 Towers of Hanoi *(continued)*

```
      {
       towers (n - 1, source, auxiliary, dest);
       printf("\t\t\t\tStep %3d: Move from %c to %c\n",
               ++step, source, dest);
       towers (n - 1, auxiliary, dest, source);
      } // if … else
    return;
}  // towers
```

The output from Program 6-27 is shown in Table 6-3.

Calls	Output
Towers (3, A, C, B)	
Towers (2, A, B, C)	
Towers (1, A, C, B)	
	Step 1: Move from A to C
	Step 2: Move from A to B
Towers (1, C, B, A)	
	Step 3: Move from C to B
	Step 4: Move from A to C
Towers (2, B, C, A)	
Towers (1, B, A, C)	
	Step 5: Move from B to A
	Step 6: Move from B to C
Towers (1, A, C, B)	
	Step 7: Move from A to C

TABLE 6-3 Tracing of Program 6-27, Towers of Hanoi

6.10 Programming Example—The Calculator Program

Let's look at our calculator program one more time. In Chapter 5, we gave users the capability of selecting one of four options: add, subtract, multiply, or divide. However, if users needed to make two calculations, they had to run the program twice. We now add a loop that allows users to make as many calculations as needed (Program 6-28). We include only two functions, *main* and `getOption`, since, at this point, all of the others are the same.

PROGRAM 6-28 The Complete Calculator

```
/* This program adds, subtracts, multiplies, and divides
   two integers.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>

// Function Declaration
int   getOption (void);
void  getData   (int* a,      int* b);
float calc      (int option,  int  num1, int num2);

float add  (float num1, float num2);
float sub  (float num1, float num2);
float mul  (float num1, float num2);
float divn (float num1, float num2);

void  printResult (float num1,   float num2,
                   float result, int   option);

int main (void)
{
// Local Declarations
   int    done = 0;
   int    option;
   float  num1;
   float  num2;
   float  result;

// Statements
   while (!done)
     {
      option = getOption();
      if (option == 5)
         done = 1;
      else
         {
          do
             {
              printf("\n\nEnter two numbers: ");
```

continued

PROGRAM 6-28 The Complete Calculator *(continued)*

```
            scanf("%f %f", &num1, &num2);
            if (option == 4 && num2 == 0)
               {
                printf("\a\n *** Error *** ");
                printf("Second number cannot be 0\n");
               } // if
           } while (option == 4 && num2 == 0);

         switch (option)
           {
            case 1: result = add (num1, num2);
                    break;
            case 2: result = sub (num1, num2);
                    break;
            case 3: result = mul (num1, num2);
                    break;
            case 4: result = divn (num1, num2);
           } // switch

         printResult (num1, num2, result, option);
        } // else option != 5
     } // while
   printf("\nThank you for using Calculator.\n");
   return 0;
}  // main

/* ==================== getOption ====================
   This function shows a menu and reads the user option.
      Pre   nothing
      Post  returns a valid option
*/
int getOption (void)
{
// Local Declarations
   int option;

// Statements
   do
     {
      printf("\n******************");
      printf("\n*      MENU      *");
      printf("\n*                *");
      printf("\n*  1. ADD        *");
      printf("\n*  2. SUBTRACT   *");
```

continued

PROGRAM 6-28 The Complete Calculator *(continued)*

```
 86         printf("\n*  3. MULTIPLY     *");
 87         printf("\n*  4. DIVIDE       *");
 88         printf("\n*  5. QUIT         *");
 89         printf("\n*                  *");
 90         printf("\n*******************");
 91
 92         printf("\n\n\nPlease type your choice ");
 93         printf("and press the return key : ");
 94         scanf("%d", &option);
 95
 96         if (option < 1 || option > 5)
 97            printf("Invalid option. Please re-enter.\n");
 98
 99        } while (option < 1 || option > 5);
100     return option;
101  }  // getOption
```

Program 6-28 Analysis As you look at the changes in this version of our program, first note the two loops in *main*. The first loop (statement 32) continues the calculator until the user says it's time to quit. The second loop (statement 39) gets and validates the numbers, making sure that the user isn't trying to divide by zero. (Your computer will get very upset if you divide by zero and will stop!)

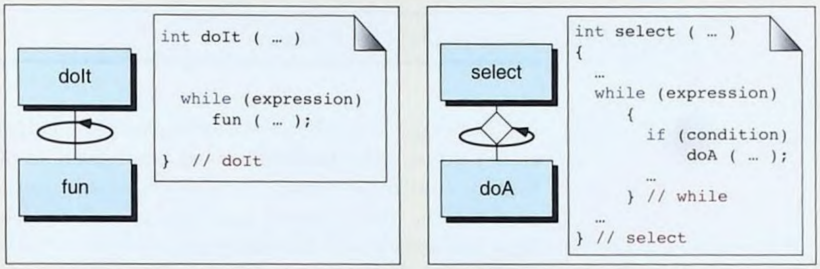
We also modified our `getOption` function to add the `quit` option and to validate the options. If the user makes a mistake, we correct it in `getOption`. Extending the concept to a general principle, whenever we write a function to get data from a user, the function should handle all data validation. This makes for much simpler code in the rest of the program.

This simplification is also seen in the *switch* statement. Since we have validated the numbers before the *switch*, we no longer need to test for a valid divisor in the fourth *case* option. We also no longer need a *default*, since we know the options are valid. The result is a simpler statement, much more in line with the KISS principle.

6.11 Software Engineering

Loops in Structure Charts

Now that you understand how to write loops, let's look at how they are shown in a structure chart. The symbols are very simple. Loops go in circles, so the symbol we use is a circle. Programmers use two basic looping symbols. The first is a simple loop. It is represented by Figure 6-31 (a). The other is the conditional loop, shown in Figure 6-31 (b).



```
int doIt ( … )


   while (expression)
      fun ( … );

} // doIt
```

```
int select ( … )
{
  …
  while (expression)
     {
       if (condition)
          doA ( … );
       …
     } // while
  …
} // select
```

(a) loop

(b) conditional loop

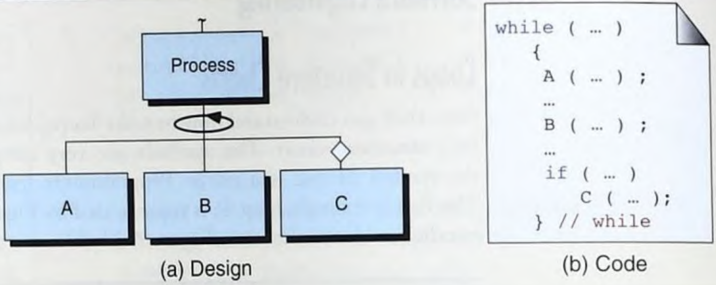
FIGURE 6-31 Structure Chart Symbols for Loops

When the function is called unconditionally, as in a *while* loop, the circle flows around the line above the called function. On the other hand, if the call is conditional, as in a function called in an *if...else* statement inside a loop, then the circle includes a decision diamond on the line.

Figure 6-32 shows the basic structure for a function called process. The circle is *below* the function that controls the loop. In this example, the looping statement is contained in process, and it calls three functions, A, B, and C. The exact nature of the loop cannot be determined from the structure chart. It could be any of the three basic looping constructs. To help you better visualize the process, however, let's further assume that the loop is a *while* loop that contains a *scanf* that reads until the end of file. Within the *while* loop are three calls: the first to A, the second to B, and the third, conditionally, to C.

Determining Algorithm Efficiency

There is seldom a single algorithm for any problem. When comparing two different algorithms that solve the same problem, we often find that one algorithm will be an order of magnitude more efficient than the other. In this case, it only makes sense that programmers have some way to recognize and choose the more efficient algorithm.



```
while ( … )
   {
    A ( … ) ;
    …
    B ( … ) ;
    …
    if ( … )
       C ( … );
   } // while
```

(a) Design

(b) Code

FIGURE 6-32 Structure Chart for `Process`

Although computer scientists have studied algorithms and algorithm efficiency extensively, the field has not been given an official name. Brassard and Bratley coined the term *algorithmics*, which they define as "the systematic study of the fundamental techniques used to design and analyse efficient algorithms."[2] We will use this term.

If a function is linear—that is, if it contains no loops—then its efficiency is a function of the number of instructions it contains. In this case, the function's efficiency depends on the speed of the computer and is generally not a factor in its overall efficiency in a program. On the other hand, functions that loop will vary widely in their efficiency. The study of algorithm efficiency is therefore largely devoted to the study of loops.

As we study specific examples, we will develop a formula that describes the algorithm's efficiency. We will then generalize the algorithm so that the efficiency can be stated as a function of the number of elements to be processed. The general format is

$$f(n) = \textbf{efficiency}$$

Linear Loops

Let's start with a simple loop. We want to know how many times the body of the loop is repeated in the following code:

```
for (int i = 1; i <= 1000; i++)
   {
    BODY;
   } // for
```

[2] Gilles Brassard and Paul Bratley, *Algorithmics Theory and Practice* (Upper Saddle River, NJ: Prentice Hall, 1988), p. xiii.

The answer is 1000 times. But the answer is not always straightforward as it is in the previous example. Consider the following loop. How many times is the body repeated in this loop? Here the answer is 500 times. Why?

```
for (int i = 1; i <= 1000; i += 2;)
    {
     BODY;
    } // for
```

In both cases, the number of iterations is directly proportionate to the limit test in the *for* loop. If we were to plot either of these loops, we would get a straight line. Thus, they are known as linear loops.

Since the efficiency is proportionate to the number of iterations, it is

$$f(n) = n$$

Logarithmic Loops

Now consider a loop in which the controlling variable is multiplied or divided in each loop. How many times will the body of the loops be repeated in the following program segments?

Multiply Loops	Divide Loops

`i =  1;`
`while (i < 1000)`
`    {`
`     BODY;`
`     i *= 2;`
`    } // while`
	`i = 1000;`
	`while (i >= 1)`
	`    {`
	`     BODY;`
	`     i /= 2;`
	`    } // while`

To help us understand this problem, Table 6-4 analyzes the values of *i* for each iteration.

Multiply		Divide	
Iteration	i	Iteration	i
1	1	1	1000
2	2	2	500
3	4	3	250
4	8	4	125

continued

TABLE 6-4 Analysis of Multiply / Divide Loops

Multiply		Divide	
Iteration	i	Iteration	i
5	16	5	62
6	32	6	31
7	64	7	15
8	128	8	7
9	256	9	3
10	512	10	1
(exit)	1024	(exit)	0

TABLE 6-4 Analysis of Multiply / Divide Loops *(continued)*

As can be seen, the number of iterations is 10 in both cases. The reason is that in each iteration the value of *i* doubles for the multiplication and is cut in half for the division. Thus, the number of iterations is a function of the multiplier or divisor; in this case, two. That is, the loop continues while the condition shown below is true.

$$\begin{array}{ll} \textbf{multiply} & \mathbf{2^{Iterations} < 1000} \\ \textbf{divide} & \mathbf{1000 / 2^{Iterations} >= 1} \end{array}$$

Generalizing the analysis, we can say that the iterations in loops that multiply or divide are determined by the following formula:

$$f(n) = ceil\ (\log_2 n)$$

Nested Loops

When we analyze loops that contain loops, we must determine how many iterations each loop completes. The total is then the product of the number of iterations for the inner loop and the number of iterations in the outer loop.

Iterations = outer loop iterations * inner loop iterations

We now look at three nested loops: linear logarithmic, dependent quadratic, and quadratic.

Linear Logarithmic

The inner loop in the following code is a loop that multiplies. (To see the multiplication, look at the update expression in the inner *for* statement.)

```
for (int i = 1; i < = 10; i ++)
        for (int j = 1; j <= 10; j *= 2)
             {
              BODY;
             } // for j
```

The number of iterations in the inner loop is therefore

$$\boldsymbol{\mathit{ceil}\ (\log_2 10)}$$

However, since the inner loop is controlled by an outer loop, the above formula must be multiplied by the number of times the outer loop executes, which is 10. This gives us

$$\mathbf{10\ (\mathit{ceil}(\log_2 10))}$$

which is generalized as

$$\boldsymbol{f(n) = n\ (\mathit{ceil}\ (\log_2 n))}$$

Dependent Quadratic

Now consider the nested loop shown in the following example:

```
for (int i = 1; i < = 10; i ++)
        for (int j = 1; j <= i; j ++)
             {
              BODY;
             } // for j
```

The outer loop is the same as the previous loop. However, the inner loop is executed only once in the first iteration, twice in the second iteration, three times in the third iteration, and so forth. The number of iterations in the body of the inner loop is mathematically stated as

$$\mathbf{1 + 2 + 3 + \ldots + 9 + 10 = 55}$$

which is generalized to

$$\mathbf{f(n) = n\frac{(n+1)}{2}}$$

Quadratic

In the final nested loop, each loop executes the same number of times, as seen in the following example:

```
for (int i = 1; i <= 10; i ++)
    for (int j = 1; j <= 10; j ++)
      {
       BODY;
      } // for j
```

The outer loop—that is, the loop at the first *for* statement—is executed ten times. For each iteration, the inner loop is also executed ten times. The answer, therefore, is 100, which is 10 * 10, the square of the loops. This formula generalizes to

$$f(n) = n^2$$

Big-O Notation

With the speed of computers today, we are not concerned with an exact measurement of an algorithm's efficiency as much as we are with its general magnitude. If the analysis of two algorithms shows that one executes 15 iterations while the other executes 25 iterations, they are both so fast that we can't see the difference. On the other hand, if one iterates 15 times and the other 1500 times, we should be concerned.

We have shown that the number of statements executed in the function for n elements of data is a function of the number of elements, expressed as $f(n)$. While the equation derived for a function may be complex, usually, a dominant factor in the equation determines the order of magnitude of the result. Therefore, we don't need to determine the complete measure of efficiency, only the factor that determines the magnitude. This factor is the big-O, as in On-the-Order-Of, and expressed as $O(n)$, that is, on-the-order-of n.

This simplification of efficiency is known as **big-O analysis**. For example, if an algorithm is quadratic, we would say its efficiency is

$$O(n^2)$$

or on-the-order of n-squared.

The big-O notation can be derived from $f(n)$ using the following steps:

1. In each term, set the coefficient of the term to 1.
2. Keep the largest term in the function, and discard the others. Terms are ranked from lowest to highest, as shown below.

$$\log n \quad n \quad n \log n \quad n^2 \quad n^3 \quad n^k \quad 2^n \quad n!$$

For example, to calculate the big-O notation for

$$f(n) = n\frac{(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$$

we first remove all coefficients. This gives us

$$n^2 + n$$

which, after removing the smaller factors, gives us

$$n^2$$

which, in Big-O notation, is stated as

$$O(f(n)) = O(n^2)$$

To consider another example, let's look at the polynomial expression

$$f(n) = a_j n^k + a_{j-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n^1 + a_0$$

We first eliminate all of the coefficients, as shown below.

$$f(n) = n^k + n^{k-1} + \cdots + n^2 + n + 1$$

The largest term in this expression is the first one, so we can say that the order of a polynomial expression is

$$O(f\,(n)) = O(n^k)$$

Standard Measures of Efficiency

Computer scientists have defined seven categories of algorithm efficiency. We list them in Table 6-5 in order of decreasing efficiency. Any measure of efficiency presumes that a sufficiently large sample is being considered. If we are only dealing with ten elements and the time required is a fraction of a second, there will be no meaningful difference between two algorithms. On the other hand, as the number of elements processed grows, the difference between algorithms can be staggering. In Table 6-5, *n* is 10,000.

Returning to the question of why we should be concerned about efficiency, consider the situation in which we can solve a problem in three ways: one is linear, another is linear-logarithmic, and the third is quadratic. The order of their efficiency for a problem containing 10,000 elements is shown in Table 6-5, along with the other algorithmics. Obviously, we wouldn't want to use the quadratic solution.

Efficiency	Big-O	Iterations
logarithmic	O(log *n*)	14
linear	O(*n*)	10,000
linear logarithmic	O(*n*(log *n*)	140,000
quadratic	$O(n^2)$	$10{,}000^2$
polynomial	$O(n^k)$	$10{,}000^k$
exponential	$O(c^n)$	$2^{10{,}000}$
factorial	O(*n*!)	10,000!

TABLE 6-5 Measures of Efficiency

Looking at the problem from the other end, if we use a computer that executes a million instructions per second and the loop contains ten instructions, then we spend .00001 second for each iteration of the loop. Table 6-5 also contains an estimate of the time to solve the problem given different efficiencies.

6.12 Tips and Common Programming Errors

1. Be aware that the *while* and *for* loops are pretest loops. Their body may never be executed. If you want your loop to be executed at least once, use a *do...while*.

2. Do not use equality and inequality for the control expression in loops; use limits that include less than or greater than. You may accidentally create an infinite loop, as shown below.

```
...
i = 0;
while (i != 13)
   {
    ...
    i++;          // sets i to 1, 3, 5, ..., 13
    ...
    i++;          // sets i to 2, 4, 6, ..., 14
    ...
   } // while
```

3. It is a compile error to omit the semicolon after the expression in the *do...while* statement.

4. It is most likely a logic error to place a semicolon after the expression in a *while* or *for* statement. (Some compilers warn you when you do.)

5. It is a compile error to code a *for* statement with commas rather than semicolons, as shown below.

```
for (int i = 0, i < 10, i++)
```

6. It is a logic error to omit the update in the body of a *while* or *do...while* loop. Without an update, either explicit or implicit, the loop never terminates.

7. It is a common logic error to miscode the limit test in *for* statements. The result is usually a loop that executes one extra time or terminates one iteration short. For example, the following statement executes nine times, not ten:

```
for (int i = 1; i < 10; i++)
```

8. It is generally a logic error to update the terminating variable in both the *for* statement and in the body of the loop, as follows.

```
for (int i = 0; i < 10; i++)
   {
    ...
    i += 1;
   } // for
```

9. A recursive function must have a base case. Therefore, it is most likely an error if a recursive function does not have an *if* statement that prevents the recursive call and allows the function to return. For example, the following code—based on Program 6-26—would never terminate.

```
long fib (long num)
{
// Statements
   return (fib (num - 1) + fib (num - 2));
}  // fib
```

6.13 Key Terms

all inquiry
any inquiry
base case
big-O analysis
body of loop
comma expression
comma operator
continue
counter-controlled loop
event-controlled loop
general case
infinite loop
initialization
inquiry
limit-test expression
loop control expression
loop update
post-test loop
pretest loop
process-control loops
recursion
summation
while

6.14 Summary

❑ The real power of computers is in their ability to repeat an operation or a series of operations many times.

❑ To control the loop, we need a condition to determine if more processing is needed.

❑ In a pretest loop, in each iteration, we check the condition first. If it is true, we iterate once more; otherwise, we exit the loop.

❑ In a post-test loop, in each iteration, we do the processing. Then we check the condition. If it is true, we start a new iteration; otherwise, we exit the loop.

❑ In a pretest loop, the processing is done zero or more times.

❑ In a post-test loop, the processing is done one or more times.

❑ In a pretest loop, if the body is executed n times, the limit test is executed $n + 1$ times.

❏ In a post-test loop, if the body is executed *n* times, the limit test is executed *n* times.

❏ The control expression in a loop must be explicitly or implicitly initialized.

❏ If you know exactly the number of times the body must be repeated, use a counter-controlled loop; if an event must occur to terminate a loop, use an event-controlled loop.

❏ C has three loop statements: *while, for,* and *do...while.*

❏ The *while* loop is a pretest loop. It can be used for a counter-controlled or event-controlled loop, but it is usually used only for event control.

❏ The *for* loop is a pretest loop. It can be used for both counter-controlled and event-controlled loops, but it is used mostly in the first case.

❏ The loop variable in a *for* loop may be locally defined in the *for* statement.

❏ The *do...while* loop is a post-test loop. It is usually used when the body must be executed at least once.

❏ We discussed two C statements that are related to looping, *break* and *continue.*

❏ The *break* statement is used to terminate a loop prematurely. We strongly recommend that you use the *break* statement only within *switch* statements.

❏ The *continue* statement is used to skip the rest of the statements in a loop and start a new iteration without terminating the loop. We strongly recommend that you never use the *continue* statement.

❏ The best loop for data validation is the *do....while* loop.

❏ Recursion is a repetitive process in which a function calls itself.

❏ The statement that solves a recursive problem is known as the base case; the rest of the function is known as the general case.

❏ A loop in a structure chart is indicated by a circle on the line connecting it to the called functions. Only loops that call other functions are shown.

6.15 Practice Sets

Review Questions

1. In a pretest loop, the limit test condition is tested first.

   a. True
   b. False

2. The action that causes the loop limit test to change from true to false is the loop update.

   a. True
   b. False

3. The value of a comma expression is the value of the first expression.

   a. True
   b. False

4. Recursion is a repetitive process in which a function calls itself.

   a. True
   b. False

5. Which of the following statements about pretest loops is true?

   a. If a pretest loop limit test is false, the loop executes one more time.
   b. Pretest loop initialization is done first in the loop body.
   c. Pretest loops execute a minimum of one time.
   d. Pretest loops test the limit condition after each execution of the loop body.
   e. The update for a pretest loop must be a part of the loop body.

6. Which of the following statements about loop initialization is false?

   a. Explicit initialization includes code to set the initial values of loop variables.
   b. Implicit initialization relies on preexisting values for loop variables.
   c. Initialization code is explicitly required in all loops.
   d. Initialization is preparation required for proper execution of a loop.
   e. Initialization must be done before the first execution of the loop.

7. Which of the following statements about loop updates is false?

   a. A loop update changes key variable(s) in a loop, thus allowing the loop to terminate.
   b. Loop updates may be made before or after a loop iteration.
   c. Loops may use explicit or implicit updates.
   d. In a *for* loop, updates are generally found in the *for* statement itself.
   e. The number of updates always equals the number of loop iterations.

8. Which of the following statements about counter-controlled loops is false?

   a. Counter-controlled loops are generally pretest loops.
   b. Counter-controlled loops generally increment or decrement a counter.
   c. Counter-controlled loops require a limit test.
   d. The number of times a loop iterates must be a constant.
   e. The update in a counter-controlled loop is generally explicit.

9. Which of the C loops is a pretest loop?

   a. *do...while*
   b. *for*
   c. *while*
   d. both the *do...while* and the *for*
   e. both the *for* and the *while*

10. Which of the following statements about the *while* statement is true?

    a. Multiple statements are allowed in a *while* loop.
    b. The limit test in a *while* loop is made before each iteration.

c. The update in a *while* statement is contained in the *while* statement expression itself.
d. The *while* statement is a post-test loop.
e. The *while* statement must be terminated with a semicolon.

11. Which of the following statements about *for* and *while* statements is false?

a. Both statements allow only one statement in the loop.
b. Both statements are pretest loops.
c. Both statements can be used for counter-controlled loops.
d. Both statements include initialization within the statement.
e. Both statements require an update statement.

12. Which of the following statements about the *do...while* loop is false?

a. A *do...while* loop executes one or more iterations.
b. Any statement may be used as the action in a *do...while*.
c. The *do...while* is best suited for use as an event-controlled loop.
d. The *do...while* is the only loop that requires a semicolon.
e. The limit test in a *do...while* loop is executed at the beginning of each iteration.

13. Nested loops have a ______________ standard measure of efficiency.

a. exponential
b. linear
c. linear logarithmic
d. quadratic
e. linear logarithmic or quadratic

14. The ______________ standard measure of efficiency is considered the most efficient.

a. exponential
b. linear
c. logarithmic
d. quadratic
e. polynomial

Exercises

15. What would be printed from each of the following program segments? Compare and contrast your answers to parts a, b, and c.

a.

```
x = 12;
while (x > 7)
   printf("%d\n", x);
```

b.

```
for (int x = 12; x > 7;)
   printf("%d\n", x);
```

c.

```
x = 12;
do
   printf("%d\n", x);
while (x > 7);
```

16. What would be printed from each of the following program segments? Compare and contrast your answers to parts a, b, and c.

a.

```
x = 12;
while (x > 7)
   {
    printf("%d\n", x);
    x--;
   }
```

b.

```
for (int x = 12; x > 7; x--)
   printf(" %d\n", x);
```

c.

```
x = 12;
   do
      {
       printf("%d\n", x);
       x--;
      } while (x > 7);
```

17. What would be printed from each of the following program segments? Compare and contrast your answers to parts a and b.

a.

```
x = 12;
while (x > 7)
   {
    printf("%d\n", x);
    x -= 2;
   }
```

b.

```
for (int x = 12; x > 7; x -= 2)
   printf ("%d\n", x);
```

18. What would be printed from each of the following program segments? Compare and contrast your answers to parts a, b, and c.

a.

```
x = 12;
while (x < 7)
   {
    printf("%d\n", x);
    x--;
   } // while
```

b.

```
for (int x = 12; x < 7; x--)
   printf ("%d\n", x);
```

c.

```
x = 12;
   do
   {
    printf("%d\n", x);
    x--;
   } while (x < 7);
```

19. Change the following *while* loops to *for* loops.

a.

```
x = 0;
while (x < 10)
   {
    printf("%d\n", x);
    x++;
   }
```

b.

```
scanf("%d", &x);
while (x != 9999)
   {
    printf("%d\n", x);
    scanf("%d", &x);
   }
```

20. Change the *while* loops in Exercise 19 to *do...while* loops.

21. Change the following *for* loops to *while* loops:

    a.

```
for (int x = 1; x < 100; x++)
   printf ("%d\n", x);
```

b.

```
for (; scanf("%d", &x) != EOF;)
   printf ("%d\n", x);
```

22. Change the *for* loops in Exercise 21 to *do...while* loops.

23. Change the following *do...while* loops to *while* loops.

    a.

```
x = 0;
do
   {
    printf("%d\n", x);
    x++;
   } while (x < 100);
```

b.

```
do
   {
    res = scanf("%d", &x);
   } while (res != EOF);
```

24. Change the *do...while* loops in Exercise 23 to *for* loops.

25. A programmer writes the following *for* loop to print the numbers 1 to 10. What is the output? If the output is incorrect, how would you correct it?

```
for (num = 0; num < 10; num++)
    printf("%d", num);
```

26. Another programmer writes the following for loop to print the numbers 1 to 10. What is the output? If the output is incorrect, how would you correct it?

```
for (int num = 0; num < 10; num++)
   {
    numOut = num + 1;
    printf("%d", numOut);
   } // for
```

27. What will be printed from the following program segments?

a.

```
for (int x = 1; x <= 20; x++)
   printf("%d\n", x);
```

b.

```
for (int x = 1; x <= 20; x++)
   {
    printf("%d\n", x);
    x++;
   } // for
```

28. What will be printed from the following program segments?

a.

```
for (int x = 20; x >= 10; x--)
   printf("%d\n", x);
```

b.

```
for (int x = 20; x >= 1; x--)
   {
    printf("%d\n", x);
    x--;
   } // for
```

29. What will be printed from the following program segments?

a.

```
for (int x = 1; x <= 20; x++)
   {
    for (int y = 1; y <= 5; y++)
       printf("%d", x);
    printf("\n");
   } // for
```

b.

```
for (int x = 20; x >= 1; x--)
   {
    for (int y = x; y >= 1; y--)
       printf("%3d", x);
    printf("\n");
   } // for
```

30. What will be printed from the following program segments?

    a.

```
for (int x = 1; x <= 20; x++)
   {
    for (int y = 1; y < x; y++)
       printf(" ");
    printf("%d\n", x);
   } // for
```

    b.

```
for (int x = 20; x >= 1; x--)
   {
    for (int y = x; y >=1; y--)
       printf (" ");
    printf ("%d\n", x);
   } // for
```

31. You find the statement shown below in a program you are maintaining.

```
for ( ; ; )
   {
    ...
   } // for
```

    a. Describe the implications behind the null expressions in the *for* statement.
    b. Since there is no limit condition, how can this statement be exited?
    c. Is this good structured programming style? Explain your answer.

Problems

32. Write a program that uses a *for* loop to print a line of 60 asterisks.

33. Write a *for* loop that will produce each of following sequences:

    a. 6, 8, 10, 12, ..., 66
    b. 7, 9, 11, 13, ..., 67
    c. The sum of the numbers between 1 and 15 inclusive
    d. The sum of the odd numbers between 15 and 45 inclusive
    e. The first 50 numbers in the series 1, 4, 7, 10, ... (calculate the total 1 + 4 + 7 + 10 + ...)

34. Write a program that prompts the user to enter an integer, n, and then n floating-point numbers. As the numbers are read, the program will calculate the average of the positive numbers.

35. Rewrite Problem 34 to average the negative numbers.

36. Write a program that asks the user to enter a list of integers. The program is to determine the largest value entered and the number of times it was entered. For example, if the following series is entered

```
5 2 15 3 7 15 8 9 5 2 15 3 7
```

it would output the largest value is 15 and it was entered 3 times.

37. Write a program that creates the following pattern:

```
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7
1 2 3 4 5 6
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

38. Write a function that creates the following pattern, given the height (number of rows) and the width (asterisks per row):

```
*************
*************
*************
*************
```

39. Write a function that creates the following pattern, given the height (number of rows) and the width (print characters per line):

```
===========
*         *
*         *
*         *
*         *
*         *
===========
```

40. Write a function that creates the following pattern, given the height (number of rows):

```
*
***
*****
*******
*********
***********
```

41. Write a function that creates the following pattern, given the height (number of rows):

```
***********
*********
*******
*****
***
*
```

42. Write a function that creates the following pattern, given the height (number of rows):

```
*
***
*****
*******
*********
*********
*******
*****
***
*
```

43. Modify Program 6-3 to display the total as each number is entered. The format should be

```
Enter your numbers: <EOF> to stop.
5
Total:  5
17
Total: 22
8
Total: 30
```

44. Write a program that reads integer data from the standard input unit and prints the minimum integer read, maximum integer read, and the average of the list. Test your program with the data shown below.

```
{24 7 31 -5 64 0 57 -23 7 63 31 15 7 -3 2 4 6}
```

45. In "The *do...while* Loop" in Section 6.5, we demonstrated the use of a *do...while* to validate input. The code fragment contains no message to tell the user that an invalid number has been entered. Write a function that reads only positive even numbers from the keyboard. If a negative or odd number is entered, it should print an error message and ask the user to enter another number. Each call to the function is to read only one

number. The valid number read is to be returned to the calling program. If the user enters `EOF`, the function should return it. Then write a short program to test the function using the data shown below. The valid numbers should be printed, either in a separate function or in *main*.

```
{2 18 -18 5 7 100 1 -1}
```

46. Write a function that reads integers from the keyboard. If any of the numbers are negative, it returns a negative number. If all the numbers are positive, it returns their average. (Hint: See Program 6-22.)

47. Program 6-19 uses a *while* loop to read a series of numbers from the keyboard. Since you will always have at least one number in this program, rewrite it to use the *do…while*.

48. Program 6-21 uses `INT_MAX` from the *limits.h* library to initialize the `smallest` variable. Another solution is to read the first number and put its value in `smallest`, then go into the loop to read the rest of the numbers. Modify Program 6-21 to make this change.

49. Euler's number, *e*, is used as the base of natural logarithms. It can be approximated using the following formula:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} + \frac{1}{6!} + \cdots + \frac{1}{(n-1)} + \frac{1}{n!}$$

Write a program that approximates *e* using a loop that terminates when the difference between two successive values of *e* differ by less than 0.0000001.

50. The value of pi (π) can be calculated by the following formula:

$$\theta = \sqrt{6\left(\frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \cdots + \frac{1}{\text{limit}^2}\right)}$$

Write a function that uses this formula to calculate pi. Then write a test driver, and run it once to test your function with a limit of 5 terms and a limit of 10 terms. Display the result of each test.

51. Write a program that reads an integer from the keyboard and then calls a recursive function to print it out in reverse. For example, if the user enters 4762, it prints 2674.

52. Rewrite Program 6-26 using an iterative solution.

Projects

53. Statisticians use many different algorithms in addition to the arithmetic average. Two other averages are the geometric and the harmonic mean.

The geometric mean of a set of n numbers, x_1, x_2, $x_3, \ldots$, x_{n-1}, x_n is defined by the following formula:

$$\sqrt[n]{x_1 \times x_2 \times \ldots \times x_n}$$

The harmonic mean is defined by the following formula:

$$\frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \cdots + \frac{1}{x_n}}$$

Write a program that reads a series of numbers and calculates the average, geometric mean, and harmonic mean.

54. Write a C program that can create four different patterns of different sizes. The size of each pattern is determined by the number of columns or rows. For example, a pattern of size 5 has 5 columns and 5 rows. Each pattern is made of character `$` and a digit, which shows the size. The size must be between 2 and 9. Table 6-6 shows the four patterns in size 5.

Pattern 1	Pattern 2	Pattern 3	Pattern 4
`5$$$$`	`$$$$5`	`$$$$$`	`$$$$$`
`$5$$$`	`$$$5$`	`$$$$5`	`5$$$$`
`$$5$$`	`$$5$$`	`$$$55`	`55$$$`
`$$$5$`	`$5$$$`	`$$555`	`555$$`
`$$$$5`	`5$$$$`	`$5555`	`5555$`

TABLE 6-6 Patterns for Problem 54

Your program displays a menu and asks the user to choose a pattern and size. But note that it must be robust; it must prompt the user to choose an option only between 1 and 5, and a pattern size only between 2 and 9. You are to print the menu and the user's response. The following example shows all user menu responses, including potential errors (user responses are in color):

```
          M  E  N  U
       1. Pattern One
       2. Pattern Two
       3. Pattern Three
```

continued

```
          4. Pattern Four
          5. Quit
Choose an Option (between 1 and 5): 11
Your Option is incorrect. Please try again.
Choose an Option (between 1 and 5): 3
Choose a Pattern Size (between 2 and 9): 12
Your Pattern size is incorrect. Try again.
Choose a Pattern Size (between 2 and 9): 4
```

The program must consist of a *main* function and six other functions called `getOption`, `getSize`, `patternOne`, `patternTwo`, `patternThree`, and `patternFour`.

Run your program *once* with the options and sizes shown in Table 6-7. Note that some options and sizes are missing because either the previous option or the size is invalid.

	Option	Size
SET 1	1	2
SET 2	2	3
SET 3	3	4
SET 4	4	5
SET 5	6	
SET 5	3	6
SET 6	2	10
SET 6		7
SET 7	5	

TABLE 6-7 Test Data for Problem 54

55. Write a C program to create a calendar for a year. The program reads the year from the keyboard. It then calculates which day of the week (SUN, MON, TUE, WED, THU, FRI, SAT) is the first day of the year and prints the calendar for that year. After printing the year, it should ask if the user wants to continue. If the answer is yes, it will print the calendar for another year until the user is done.

    The program prompts the user for the input, as shown in the next example.

```
Enter the year for your calendar :    2000
```

The output is a calendar for the whole year (12 months). One month is shown in the next example.

```
JANUARY                       2000
SUN MON TUE WED THU FRI SAT
                              1
  2   3   4   5   6   7   8
  9  10  11  12  13  14  15
 16  17  18  19  20  21  22
 23  24  25  26  27  28  29
 30  31
```

To print the correct calendar for the requested year, you must first find which day of the week is the first day of the that year. This can be done with the following formula. (For a complete explanation, see Chapter 5, Project 61.)

$$\left(((\mathbf{year}-1)\times 365)+\left[\frac{\mathbf{year}-1}{4}\right]-\left[\frac{\mathbf{year}-1}{100}\right]+\left[\frac{\mathbf{year}-1}{400}\right]+1\right)\%\ 7$$

You also must calculate leap years. The formula for this calculation is also found in Chapter 5, Project 61.

Run your program once with the following sets of data[3]:

```
SET 1: 2005
SET 2:    0
SET 3: 2000
SET 4:  123
```

56. Write a C program to help a prospective borrower calculate the monthly payment for a loan. The program also prints the amortization (payoff) table to show the balance of the loan after each monthly payment.

The program prompts the user for input, as shown in the following example:

```
Amount of the loan (Principal)? 10000.00
Interest rate per year (per cent)? 12
Number of years? 10
```

The program then creates an information summary and amortization table.

[3] The Julian calendar was changed to our current (Gregorian) calendar in 1752. Although calendars before this date are valid Gregorian calendars, they do not represent the Julian calendar in use at that time.

Banks and financial institutions use different formulas to calculate the monthly payment of a loan. For the purpose of this assignment, we use a simple formula

```
NM      = (NY * 12)
IM      = (IY / 12) / 100
P       = $(1 + IM)^{NM}$
Q       = (P / (P - 1))

MP      = (PR * IM * Q)
```

where

NY:	Scheduled number of years to amortize the loan
NM:	Scheduled number of months for the loan
IY:	Interest rate per year (as a percentage)
IM:	Interest rate/month (decimal)
PR:	Principal (the amount of the loan)
P:	The value of $(1 + IM)^{NM}$
Q:	The value of $P / (P - 1)$
MP:	Monthly payment

main must call three other functions: `calculateMonthlyPayment`, `printInformation`, and `printAmortizationTable`. Of course, the first function may also call other functions if necessary.

Because of the approximation used in calculation, the value of the new balance at the end of the last month may become nonzero. To prevent this, the last payment must be adjusted. It may be less or greater than the other months. It must be calculated by adding the principal paid to the interest paid for that month. The new balance at the end of the last month must be zero.

The following example shows the concept. The program has been run for a loan of $5000.00 at 11% interest rate for a period of 1 year. The input was

```
Amount of the loan (Principal)?  5000.00
Interest rate / year (per cent)? 11
Number of years? 1
```

Run your program once with the test data shown in Table 6-8.

	Amount	Interest	Years
Set 1	10,000.00	12	1
Set 2	5,000.00	10	2
Set 3	1,000.00	8	3

TABLE 6-8 Test data for Problem 56

The output is shown in Table 6-9. Note: Your answer may look a little different (a few pennies) because of different precision.

```
  The amount of the loan (principal):         5000.00

  Interest rate/year (percent):                  11.0

  Interest rate/month (decimal):                   0.009167

  Number of years:                                 1

  Number of months:                               12

  Monthly payment:                               441.91

         Old      Monthly  Interest Principal    New
Month  Balance    Payment    Paid      Paid    Balance

  1    5000.00    441.91    45.83    396.08   4603.92

  2    4603.92    441.91    42.20    399.71   4204.21

  3    4204.21    441.91    38.54    403.37   3800.84

  4    3800.84    441.91    34.84    407.07   3393.77

  5    3393.77    441.91    31.11    410.80   2982.97

  6    2982.97    441.91    27.34    414.57   2568.40

  7    2568.40    441.91    23.54    418.37   2150.03

  8    2150.03    441.91    19.71    422.20   1727.83

  9    1727.83    441.91    15.84    426.07   1301.76

 10    1301.76    441.91    11.93    429.98    871.78

 11     871.78    441.91     7.99    433.92    437.86

 12     437.86    441.87     4.01    437.86      0.00

 Total amount paid: 5302.88
```

TABLE 6-9 Sample Output from Project 57

57. Write a program that reads a list of integers from the keyboard and creates the following information:

a. Finds and prints the sum and the average of the integers
b. Finds and prints the largest and the smallest integer
c. Prints a Boolean (true or false) if some of them are less than 20
d. Prints a Boolean (true or false) if all of them are between 10 and 90

The input data consist of a list of integers with a sentinel. The program must prompt the user to enter the integers, one by one, and enter the sentinel when the end of the list has been reached. The prompt should look like the following:

```
Enter numbers with <return> (99999 to stop):
```

The output should be formatted as shown below.

```
The number of integers is:              xxx
The sum of the integers is:            xxxx
The average of the integers is:          xxx.xx
The smallest integer is:                xxx
The largest integer is:                 xxx
At least one number was < 20:        <true or false>
All numbers were (10 <= n >= 90): <true or false>\
```

58. The formula for converting centigrade temperatures to Fahrenheit is

$$\mathbf{F} = 32 + \left(\mathbf{C} \times \frac{\mathbf{180.0}}{\mathbf{100.0}}\right)$$

Write a program that prints out conversion tables for Celsius to Fahrenheit (0° to 100°) and Fahrenheit to Celsius (32° to 212°). Use separate functions to convert Fahrenheit to Celsius and Celsius to Fahrenheit. The output format for each table is to fit on a standard monitor display, 80 columns by 20 rows.

59. Rewrite "Towers of Hanoi" Program 6-27 using an iterative solution.

Chapter 7
Text Input/Output

A program is a data processor: It accepts input data, processes data, and creates output data. Handling input/output is a complex task, largely because of the variety of devices and data formats. Data may come from many different sources and may go to different destinations. For example, data may come from such diverse sources as a keyboard, a file on the disk, a heating or air-conditioning system, or a communication channel (network or Internet). Data may also go to many destinations, such as a monitor or a file on the disk, or to a communication channel. In this chapter, we concentrate on the keyboard and files as input entities and the monitor and files as output entities.

Objectives

- ❑ To understand the basic properties and characteristics of external files
- ❑ To understand the C implementation of file I/O using streams
- ❑ To write programs that read and write text files using the formatting functions
- ❑ To write programs that read and write text files using the C character I/O functions
- ❑ To write programs that handle simple I/O errors
- ❑ To understand and implement basic data validation concepts

7.1 Files

A **file** is an external collection of related data treated as a unit. The primary purpose of a file is to keep a record of data. Since the contents of primary memory are lost when the computer is shut down, we need files to store our data in a more permanent form. Additionally, the collection of data is often too large to reside entirely in main memory at one time. Therefore, we must have the ability to read and write portions of the data while the rest remain in the file.

A **file** is an external collection of related data treated as a unit.

Files are stored in **auxiliary** or **secondary storage devices**. The two most common forms of secondary storage are disk (hard disk, CD, and DVD) and tape.

When the computer reads, the data move from the external device to memory; when it writes, the data move from memory to the external device. This data movement often uses a special work area known as a buffer. A **buffer** is a temporary storage area that holds data while they are being transferred to or from memory. The primary purpose of a buffer is to synchronize the physical devices with a program's needs, especially on large storage devices such as disk and tape. Because of the physical requirements of these devices, more data can be input at one time than a program can use. The buffer holds the extra data until the program or the user is ready for it. Conversely, the buffer collects data until there are enough to write efficiently. These buffering activities are taken care of by software known as device drivers or access methods provided by the supplier of the computer's operating system.

As a file is being read, there eventually comes a point when all the data have been input. At this point, the file is said to be at **end of file**. The end of file in an auxiliary device is detected automatically by the device and passed to the program. It is the programmer's job to test for end of file. This test is often done in a loop control statement.

File Name

Every operating system uses a set of rules for naming its files. When we want to read or write auxiliary storage files, therefore, we must use the operating systems rules when we name the files. We refer to the operating system's name as the *file name* in this text.

File Information Table

A program that reads or write files needs to know several pieces of information, such as the operating system's name for the file, the position of the current character in the file, and so on. C has predefined a file structure[1] to hold this information. The *stdio.h* header file defines the file structure; its identifier is `FILE`. When we need a file in our program, we declare it using the `FILE` type.

7.2 Streams

Although the source and destination of data is a file or a physical device in C, as we briefly discussed in Chapter 2, data are input to and output from a stream. A stream can be associated with a physical device, such as a terminal, or with a file stored in auxiliary memory. Figure 7-1 shows this idea.

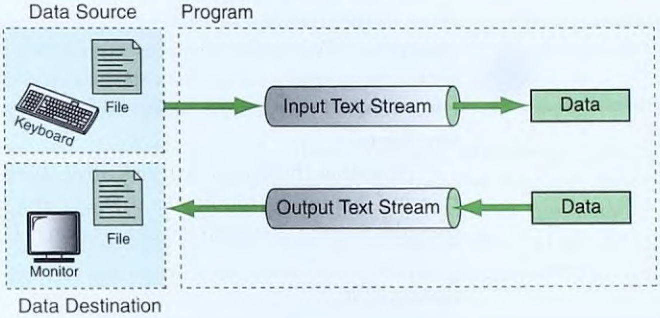


FIGURE 7-1 Streams

Text And Binary Streams

C uses two types of streams: text and binary. A text stream consists of a sequence of characters divided into lines with each line terminated by a newline (\n). A binary stream consists of a sequence of data values such as integer, real, or complex using their memory representation. In this chapter, we discuss only text streams; binary streams are discussed in Chapter 13.

Stream-File Processing

A file exists as an independent entity with a name known to the operating system. A stream is an entity created by the program. To use a file in our program, we must associate the program's stream name with the operating system's file name.

In general, there are four steps to processing a file. We first create the stream. We then open the file, which associates the stream name with the file

[1] Technically, a structure (*struct* type). We discuss structures in Chapter 12.

name. Once the file is opened, we read or write data; that is, we process the file. When the processing is complete, we close the file. We describe the steps to make this association in this section.

Creating a Stream

We create a stream when we declare it. The declaration uses the `FILE` type as shown in the following example. The `FILE` type is a structure that contains the information needed for reading and writing a file.

```
FILE* spData;
```

In this example, `spData` is a pointer to the stream. Because of the importance of streams in a program, we provide a special notation to help us recognize them. The *sp* in the name stands for stream-pointer. We then follow it with a descriptive name for the data that flows through the stream. In this case we used the generic name, *data*. In our examples, we use more descriptive names.

Note that there is an asterisk after the file type (`FILE`). We saw this notation in Chapter 4. It indicates that `spData` is a pointer variable that contains the address of the stream we are creating. It is an error to omit the asterisk.

Opening a File

Once the stream has been created, we are ready to associate the stream to a file. This is done, as we discuss in detail in the next section, through the standard open function. When the file is opened, the stream and the file are associated with each other, and the `FILE` type is filled with the pertinent file information. The open function returns the address of the file type, which is stored in the stream pointer variable, `spData`. The file open function creates the stream, which we then refer to by its name.

Using the Stream Name

After we create the stream, we can use the stream pointer in all functions that need to access the corresponding file for input or output. For example, a function can use the stream pointer to read from the file through the corresponding stream.

Closing the Stream

When the file processing is complete, we close the file. Closing the file breaks the association between the stream name and the file name. After the close, the stream is no longer available and any attempt to use it results in an error. To close the association, we need to use a close function to release the file.

System-Created Streams

We discussed that a terminal—that is, the keyboard or monitor—can be the source or destination of a text stream. C provides standard streams to communicate with a terminal. These streams must be created and associated with their terminal devices just like files. The difference is that C does it automatically for us.

C declares and defines three stream pointers in the *stdio.h* header file. The first, *stdin,* points to the standard input stream; the second *stdout*, points to the standard output stream; and the third, *stderr*, points to the standard error stream. Note that these streams are created when the program starts and we cannot declare them.

Standard stream names have already been declared in the stdio.h header file and cannot be declared again in our program.

The association between the three standard streams and the keyboard and the monitor is also done automatically when the program starts. Therefore, we cannot open any of the standard streams in our code. Like file streams, the standard streams must be closed at the end of the program. However, they are closed automatically when the program terminates.

There is no need to open and close the standard streams. It is done automatically by the operating system.

Using Stream Names

C includes many standard functions to input data from the keyboard and output data to the monitor automatically without the need for explicitly using these standard streams. For these functions, the stream that connects our programs to the terminal is automatically created by the system and we do not have to do anything more. Examples of such functions are *scanf* and *printf*, which we defined and used in Chapter 2.

7.3 Standard Library Input/Output Functions

The *stdio.h* header file contains several different input/output function declarations. They are grouped into eight different categories, as shown in Figure 7-2. The first three will be discussed in the following sections. Those shown in shaded boxes will be discussed in Chapters 11 and 13.

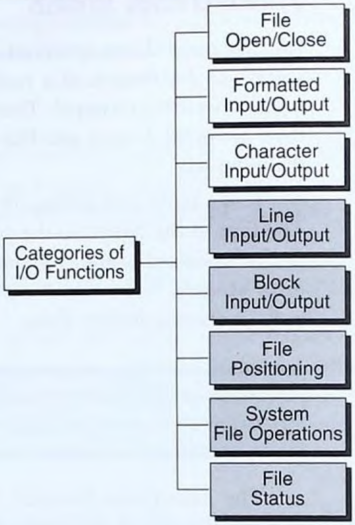


FIGURE 7-2 Categories of Standard Input/Output Functions

File Open and Close

In this section we discuss the C functions to open and close streams.

File Open (*fopen*)

The function that prepares a file for processing is *fopen*. It does two things: First, it makes the connection between the physical file and the file stream in the program. Second, it creates a program file structure to store the information needed to process the file.

To open a file, we need to specify the physical filename and its mode, as shown in the following statement.

```
fopen("filename", "mode");
```

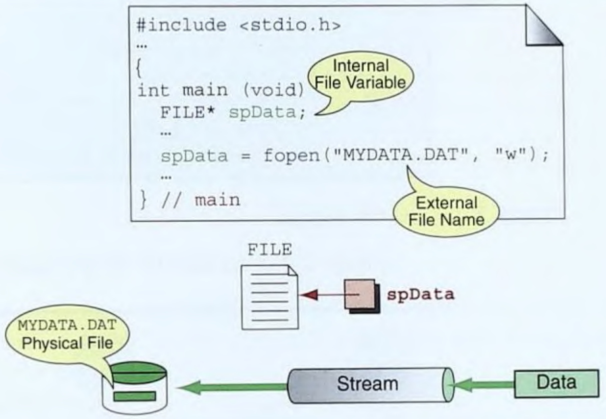
Let's examine this statement by starting on the right. The **file mode** is a string that tells C how we intend to use the file: Are we going to read an existing file, write a new file, or append to a file? We discuss file modes later in this section.

A **filename** is a string that supplies the name of the physical file as it is known to the external world. For example, when we work in a Microsoft Windows system, the file name consists of a name, a dot (period), and a three-character file extension.

The address of the file structure that contains the file information is returned by *fopen*. The actual contents of FILE are hidden from our view because we do not need to see them. All we need to know is that we can store the address of the file structure and use it to read or write the file. A complete open statement is shown below. Continuing with our temperatures example, we could open it for output (writing) with the following open statements. The first is the basic format as it might be used for the current directory in UNIX or Windows; the second is the Windows version to open a file for drive A.[2]

```
spData = fopen("MYFILE.DAT", "w");
spData = fopen("A:\\MYFILE.DAT", "w");
```

We see this open statement in the program in Figure 7-3. Some explanation of the program segment in the figure is in order. You should first note that the standard input/output library is called out at the beginning of the program.



```
#include <stdio.h>
...
{
int main (void)
  FILE* spData;
  ...
  spData = fopen("MYDATA.DAT", "w");
  ...
} // main
```

FIGURE 7-3 File Open Results

Next, note the name we have used for the file address. The sp stands for "stream pointer"; we use it in all of our stream identifiers. To the stream pointer abbreviation we add the name of the file. Because this is a generic example, we use "Data." This combination gives a readable name that is easy to remember.

Our stream pointer variable, spData, is assigned the address of the file structure when we open the file; the return value from the *fopen* function contains the file structure address. Later in the program, when we need to read or write the file, we use this pointer.

[2] Note the two backslashes in the filename. Because the backslash is the escape character, to get one in the output, we must code two.

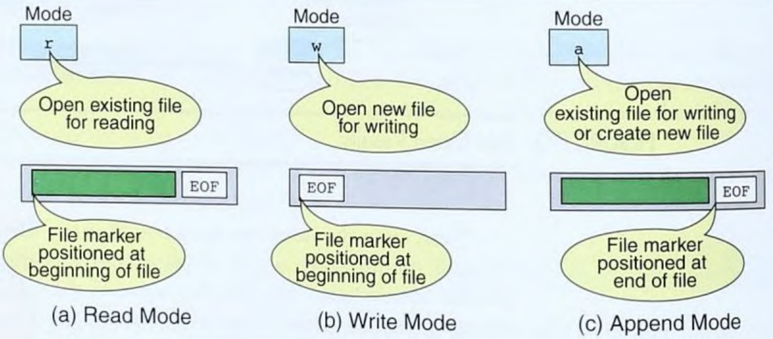
File Modes

When we open a file, we explicitly define its mode. The mode shows how we will use the file: for reading, for writing, or for appending, which means adding new data at the end of the current file. C has six different file modes. The first three, which we discuss here, are used to read, write, or append text files. In Chapter 13 we discuss the other modes, which allow both reading and writing in the same file. The mode codes discussed in this chapter are shown in Table 7-1.[3]

Mode	Meaning
r	Open text file in read mode • If file exists, the marker is positioned at beginning. • If file doesn't exist, error returned.
w	Open text file in write mode • If file exists, it is erased. • If file doesn't exist, it is created.
a	Open text file in append mode • If file exists, the marker is positioned at end. • If file doesn't exist, it is created.

TABLE 7-1 Text File Modes

Figure 7-4 describes the simple open modes.


Mode
r
Open existing file for reading
EOF
File marker positioned at beginning of file
Mode
w
Open new file for writing
EOF
File marker positioned at beginning of file
Mode
a
Open existing file for writing or create new file
EOF
File marker positioned at end of file

(a) Read Mode (b) Write Mode (c) Append Mode

FIGURE 7-4 File-Opening Modes

Read Mode

The **read mode** (`r`) opens an existing file for reading. When a file is opened in this mode, the file marker is positioned at the beginning of the file (the first

[3] There are more modes. We discuss them in Chapter 13.

character). The file marker is a logical element in the file structure that keeps track of our current position in the file.

The file must already exist: If it does not, NULL is returned as an error. Files opened for reading are shown in Figure 7-4a. If we try to write to a file opened in the read mode, we get an error message.

Write Mode

The **write mode** (w) opens a file for writing. If the file doesn't exist, it is created. If it already exists, it is opened and all its data are deleted; that is, it assumes the status of an empty file. It is an error to try to read from a file opened in write mode. A file opened for writing is shown in Figure 7-4b.

Append Mode

The **append mode** (a) also opens an existing file for writing. Instead of creating a new file, however, the writing starts after the last character; that is, new data are added, or *appended,* at the end of the file.

If the file doesn't exist, it is created and opened. In this case, the writing will start at the beginning of the file; the result will be logically the same as opening a new file for writing. Files opened in append mode are shown in Figure 7-4c. If we try to read a file opened for write append, we get an error message.

File Close (*fclose*)

When we no longer need a file, we should close it to free system resources, such as buffer space.[4] A file is closed using the close function, *fclose,* as shown in the following section.

```
#include <stdio.h>
…
int main (void)
{
// Local Declarations
   FILE* spTemp;

// Statements
   …
   spTemp = fopen("MYDATA.DAT", "w");
   …
   fclose(spTemps);
   …
```

[4] Some operating systems allow a file to be opened by only one application program at a time. Closing the file also allows other application programs to have access to the file.

Open and Close Errors

What if the open or close fails? Open and close errors occur for a number of reasons. One of the most common errors occurs when the external filename in the open function call does not match a name on the disk. When we create a new file, the open fails if there isn't enough room on the disk.

Always check to make sure that a stream has opened successfully. If it has, then we have a valid address in the file variable. But if it failed for any reason, the stream pointer variable contains `NULL`, which is a C-defined constant for no address in *stdio.h*.

Similarly, we can test the return value from the close to make sure it succeeded. The *fclose* function returns an integer that is zero if the close succeeds and `EOF` if there is an error. `EOF` is defined in the standard input/output header file. Traditionally, it is –1, but the standard defines it as any noncharacter value. To ensure that the file opened or closed successfully, we use the function as an expression in an *if* statement, as shown in Program 7-1.

PROGRAM 7-1 Testing for Open and Close Errors

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

int main (void)
{
// Local Declarations
FILE* spTemps;

// Statements
   ...

   if ((spTemps = fopen("TEMPS.DAT", "r")) == NULL)
      {
       printf("\aERROR opening TEMPS.DAT\n");
       exit (100);
      } // if open
   ...

   if (fclose(spTemps) == EOF)
      {
       printf("\aERROR closing TEMPS.DAT\n");
       exit (102);
      } // if close
   ...

}  // main
```

Program 7-1 Analysis

The most common mistake in testing for a successful open is getting the parentheses wrong, as shown below.

```
if (spTemps = fopen("TEMPS.DAT", "w") == NULL)
```

The above statement is syntactically correct (we do not get a compile error) but invalid. You must test the address returned by *fopen* after it has been assigned to `spTemps`. In the above statement, `spTemps` is assigned the logical value of the expression

```
fopen("TEMPS.DAT", "w") == NULL
```

because the equal operator has a higher precedence than the assignment operator. Study the open in Program 7-1 carefully, and make sure you understand the difference between it and the incorrect version shown in the previous example.

The error testing for the close is much simpler; we can use a simple test for the error code, `EOF`. Note that we have assigned a different return code to distinguish the open failure from the close failure.

7.4 Formatting Input/Output Functions

In Chapter 2 we introduced two formatting input/output functions, *scanf* and *printf*. The *scanf* function receives a text stream from the keyboard and converts it to data values to be stored in variables. The *printf* function receives data values from the program and converts them into text stream to be displayed on the monitor.

These two functions can be used only with the keyboard and monitor. The C library defines two more general functions, *fscanf* and *fprintf*, that can be used with any text stream. Table 7-2 compares these four input/output functions.

Terminal Input/Output

`scanf ("control string", ...);`
`printf("control string", ...);`

General Input/Output

`fscanf (stream_pointer, "control string", ...);`
`fprintf(stream_pointer, "control string", ...);`

TABLE 7-2 Formatting Functions

Stream Pointer

The first argument in the text input/output function, stream pointer, is the pointer to the streams that has been declared and associated with a text file. The following example demonstrates the use of an input stream.

```
FILE* spIn;
...
spIn = fopen("file name", "r");
...
fscanf(spIn, "format string", address list);
```

While we normally read data from the terminal keyboard using *scanf*, we can also use *fscanf*. When we use *fscanf* we must specify that the stream pointer is *stdin*, as shown in the following example.

```
fscanf(stdin, "format string", address list);
```

The following example demonstrates the use of an output stream.

```
FILE* spOut;
...
spOut = fopen("file name", "w");
...
fprintf(spOut, "format string", address list);
```

Similarly, we can use *fprintf* to print to the terminal monitor by specifying that the stream is *stdout*, as shown in the following example.

```
fprintf(stdout, "format string", value list);
```

Format Control Strings

Input and output functions for text files use a format string to describe how data are to be formatted when read or written. The format control string consists of three types of data, which may be repeated: whitespace, text characters, and the most important of the three, the conversion specification that describes how the data are to be formatted as they are read or written.

Whitespace

Format control string whitespace is handled differently for input and output. In an input function, one or more whitespaces in the format string cause zero, one, or more whitespaces in the input stream to be read and discarded. Thus, any sequence of consecutive whitespace characters in the format string will match any sequence of consecutive whitespace characters, possibly of different length, in the input stream.

A whitespace character in an input format string causes leading whitespace characters in the input to be discarded. A whitespace character in an output format string is copied to the output stream.

Whitespace in an output function is simply copied to the output stream. Thus, a space character is placed in the output stream for every space character in the format string. Likewise, tabs in the format string are copied to the output stream. This is not a good idea, however, because we can't see tabs in the format string. It is better to use the tab escape character (`\t`) so that we can see the tabs.

Text

Any text character other than a whitespace in an input format string must match exactly the next character of the input stream. If it does not match, a conflict occurs that causes the operation to be terminated. The conflicting input character remains in the input stream to be read by the next input operation on that stream. We therefore recommend that you do not use text characters in an input format string.

Text characters in an output format string are copied to the output stream. They are usually used to display messages to the user or to label data being output.

Conversion Specification

The conversion specification consists of a percent character (`%`), optional formatting instructions, and a conversion code. With one exception, each conversion specification must have a matching parameter in the parameter list that follows the format string. The type in the conversion specification and the type of the parameter must match.

The number, order, and type of the conversion specifications must match the number, order, and type of the parameters in the list. Otherwise, the result will be unpredictable and may terminate the input/output function.

Conversion specifications can have up to six elements, as shown in Figure 7-5. (Note that for input there are only five; precision is not allowed.) The first element is a conversion specification token (`%`). The last element is the conversion code. Both of these elements are required; the other elements are optional. Generally, the meaning and usage of each element is the same for both input and output. The exceptions are noted in the following discussion.

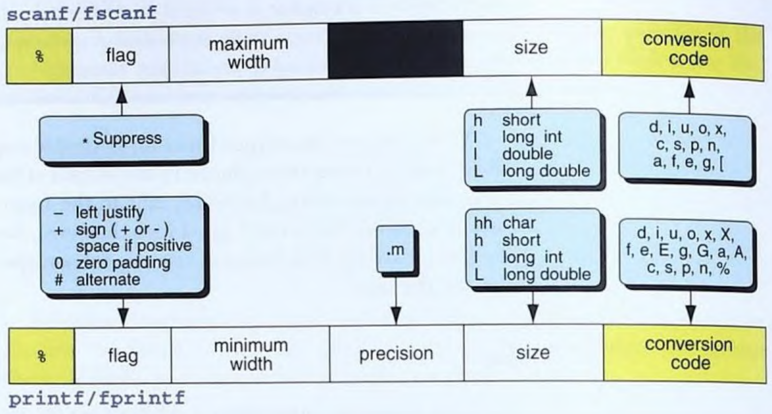


FIGURE 7-5 Conversion Specifications

Conversion Codes

The conversion code specifies the type of data that are being formatted. For input, it specifies the type of variable into which the formatted data are stored. For output, it specifies the type of data in the parameter associated with the specification. *It is our responsibility to ensure that the data are of the right type.* If they are not, strange formatting may result. Conversion codes are discussed in detail in the "Input Formatting (*scanf* and *fscanf*)" and "Output Formatting (*printf* and *fprintf*)" sections that follow.

Input Formatting *(scanf* and *fscanf)*

The *scanf* and *fscanf* functions read text data and convert the data to the types specified by a format string. The only difference between them is that *scanf* reads data from the standard input unit (the keyboard by default) and *fscanf* reads the input from a file specified by the first parameter. This file can be standard input (*stdin*).

scanf reads from stdin; fscanf reads from a user-specified stream.

The name *scanf* stands for "scan formatted"; the name *fscanf* stands for "file scan formatted." These functions have the following formats:

```
scanf ("format string", address list);
fscanf(sp, "format string", address list);
```

where sp is the address of a stream defined as type `FILE*`, `"format string"` is a string containing formatting instructions, and the address list specifies the addresses where the data are to be stored after they have been formatted. A comma must separate the format string from the variable list. If more than one variable is present, then separate the variables from each other by commas.

We must include a variable address in the address list for every conversion specification in the format string that requires data. If we do not, the result is "unpredictable and undefined." This is a standard disclaimer that means anything can happen. The usual result is that the program doesn't do what we expect and often crashes.

Input Data Formatting

The conversion operation processes input characters until any of the following occur:

1. End of file is reached.
2. An inappropriate character is encountered.
3. The number of characters read is equal to an explicitly specified maximum field width.

Input Conversion Specification

In Chapter 2 we introduced some of the conversion specifications. We discuss the rest here. Table 7-3 shows the type of the pointer argument, size, and conversion codes for *scanf* family. Note that we have not shown a flag specification in the table, because there is only one flag, suppress (*), which can be used with all input format codes.

Argument Type	Size Specifier	Code
integral	hh (char), h (short), none (int), l (long), ll (long long)	i
integer	h (short), none (int), l (long). ll (long long)	d
unsigned int	hh (char), h (short), none (int), l (long), ll (long long)	u
character octal	hh (unsigned char)	o
integer hexadecimal	h (short), none (int), l (long), ll (long long)	x
real	none (float), l (double), L (long double)	f
real (scientific)	none (float), l (double), L (long double)	e
real (scientific)	none (float), l (double), L (long double)	g
real (hexadecimal)	none (float), l (double), L (long double)	a

continued

TABLE 7-3 Sizes and Conversion Code for *scanf* Family

Argument Type	Size Specifier	Code
character	none (char), l (wchar_t)	c
string	none (char string), l (wchar_t string)	s
pointer		p
integer (for count)	none (int), hh (char), h (short), l (long), ll (long long)	n
set	none (char), l (wchar_t)	[

TABLE 7-3 Sizes and Conversion Code for *scanf* Family *(continued)*

Flag

There is only one flag for input formatting, the assignment suppression flag (*). More commonly associated with text files, the assignment suppression flag tells *scanf* that the next input field is to be read but not stored. It is discarded. The following *scanf* statement reads an integer, a character, and a floating-point number from the input stream. The character is read and discarded. The other fields are read, formatted, and stored. Note that there is no matching address parameter for the data to be discarded.

```
scanf ("%d %*c %f", &x, &y);
```

Width

The **width** specifies the maximum width of the input (in characters). This allows us to break out a code that may be stored in the input without spaces. Consider what happens when we read a Social Security number from a file and the number has no formatting; it is just nine digits in a row, followed by a space. We could read it into three variables, thus allowing us to format it with dashes in our output, with the following format specifications:

```
scanf("%3d%2d%4d…", &ssn1, &ssn2, &ssn3, …);
```

Note that the width is a maximum. If the amount of data is less than required, the scan terminates. What determines the end of the data depends on the type of data, but generally whitespace will terminate most scans.

Size

The **size** specification is a modifier for the conversion code. Used in combination with the conversion code, it specifies the type of the associated variable, for example, a *long double* (`Lf`). The size codes, along with their associated conversion codes, are explained in Table 7-3, "Sizes and Conversion Code for *scanf* Family."

Conversion Codes

With the exception of string and pointer, this section discusses the input conversion codes. The pointer conversion code is discussed in Chapter 9; the string conversion code is discussed in Chapter 11.

Integer (d) The decimal (d) format code accepts a value from the input stream and formats it into the specified variables. It reads only decimal digits and an optional plus or minus sign as the first character of the value.

Integer (i) The integer format code (i) allows the user to key a decimal, octal, or hexadecimal number. Numbers starting with any digit other than zero are read and stored as decimal values. Numbers starting with zero are interpreted as octal values and are converted to decimal and stored. Hexadecimal numbers must be prefixed with 0x or 0X; the hexadecimal value is converted to decimal and stored. We strongly recommend that the integer format code be used only when non-decimal data is being entered—such as when programmers are entering technical data; in all other cases, the decimal format code (d) should be used.

Octal and Hexadecimal (o, x) The octal (o) and hexadecimal (x) conversion codes perform unsigned conversion. For octal, the only valid input digits are 0...7. For hexadecimal input, the valid digits are 0...9, a...f, and A...F. If you are not familiar with octal or hexadecimal numbers, see Appendix D, "Numbering Systems."

Scientific Notation (e, g, a) The C language uses three real format codes for scientific notation. In scientific notation, the significand and exponent are specified separately. The significand part is a floating-point number that contains as many significant digits as possible. For example, if it contains six digits, then the number is significant only to six digits; if it has twelve digits, then it is significant to twelve digits. The larger the significance, the greater the precision. Therefore, *long double* may be more precise than *double,* which may be more precise than *float*.

The exponent specifies the magnitude of the number. It may be either positive or negative. If it is positive, then the number is the significand times ten to the power of the exponent, which may be a very large number. If it is negative, then the number is the significant times the reciprocal of the base ten exponent, which may be a very small number. These forms are shown below.

$$123e03 \rightarrow 123*10^{3} \qquad 123e-03 \rightarrow 123*10^{-3}$$

All of the following numbers are in scientific notation:

```
3e     -1.0e-3     0.1+e1     2e2
```

For input, all conversion codes (f, e, g, a) accept all three scientific notations as well as algebraic notation.

Count (n) To verify the number of input characters, we use the n conversion code. This code requires a matching variable address into which *scanf* places the count of the characters input. In the following example, count is a short integer that is to receive the number of characters read:

```
scanf("%d %8.2f %d %hn", &i, &x, &j, &count);
```

Input Side Effect and Value

When the *scanf* function is called, it can create a side effect and return a value. The side effect is to read characters from a file and format them into variables supplied in the parameter list. It continues until the end of the line is reached or until there is a conflict between the format string and a character read from the input stream. In either case, the scan returns the *number of successfully formatted values* before termination. If the scan detects the end-of-file before any conflict or assignment is performed, then the function returns EOF. The difference between the side effect and the value is seen in Figure 7-6.

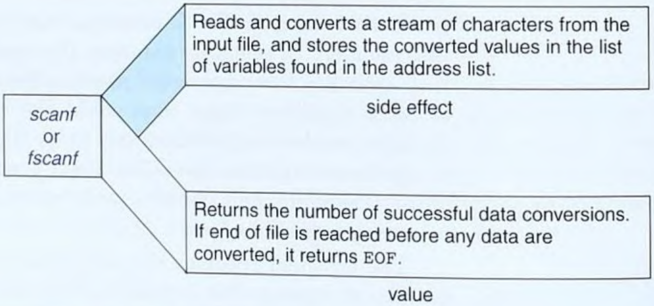
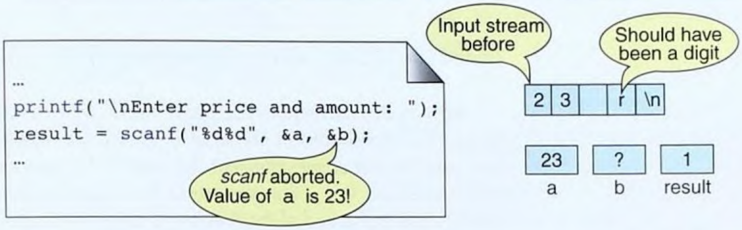

Reads and converts a stream of characters from the input file, and stores the converted values in the list of variables found in the address list.
side effect
scanf or fscanf
Returns the number of successful data conversions. If end of file is reached before any data are converted, it returns EOF.
value

FIGURE 7-6 Side Effect and Value of *scanf* and *fscanf*

This leads us to a common example of a **value error**. Suppose we want to read two numbers with one scan statement. If we succeed, the scan function returns a 2, indicating that both were read correctly. If the user makes a mistake with the first number, the scan function returns a 0. If there is a mistake with the second entry, *scanf* returns a 1. This allows us to validate that at least the correct amount and type of data were read. An error reading the second piece of data is shown in Figure 7-7.



```
…
printf("\nEnter price and amount: ");
result = scanf("%d%d", &a, &b);
…
```

FIGURE 7-7 Another Common Error

When we read data, we should verify that the data were read correctly. This is easily done by checking the return value from the scan function. Program 7-2 is a program fragment that shows how the two numbers in Figure 7-7 could be checked.

PROGRAM 7-2 Checking *scanf* Results

```
#define FLUSH while (getchar() != '\n')
#define ERR1 "\aPrice incorrect. Re-enter both fields\n"
#define ERR2 "\aAmount incorrect. Re-enter both fields\n"

// Read price and amount
    do
      {
       printf("\nEnter amount and price: ");
       ioResult = scanf("%d%f", &amount, &price);

       if (ioResult != 2)
          {
           FLUSH;
           if (ioResult == 1)
               printf(ERR1);
           else
               printf(ERR2);
          } // if
      } while (ioResult != 2);
```

```
Results:
Enter amount and price: ? 15.25
Amount incorrect. Re-enter both fields

Enter amount and price: 100 ?
Price incorrect. Re-enter both fields

Enter amount and price: 100 15.25
```

Program 7-2 Analysis We have introduced a new define statement, `FLUSH`, in this program segment.

```
#define FLUSH while (getchar( ) != '\n')
```

Why is this statement necessary? Recall that when *scanf* encounters an error, it leaves the invalid data in the input stream. If we simply print the user messages and then return to the *scanf* statement, the invalid data are still there. We need to get rid of them somehow! This is the purpose of the `FLUSH` statement. It reads all data from the invalid input to the end of the line. For a complete discussion of the flush statement, see "Testing for Valid User Input" in the software engineering section.

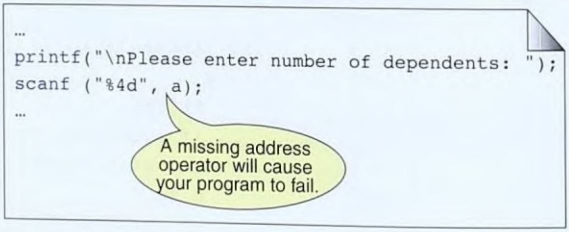
Also note how the error messages tell the user exactly what went wrong, rather than giving a general "The numbers entered are incorrect" type of message. Whenever possible, you should tell the user exactly what went wrong.

Two Common Input Format Mistakes

There are as many mistakes to be made as there are programmers programming. Two are so common that they need to be emphasized.

Invalid Address

The first common mistake is to use a data value rather than an address for the input parameter. The scan function places the formatted input at the address specified in the parameter list. When we pass data rather than an address, C interprets the data as an address. This causes a part of memory to be destroyed. If we are lucky, our program fails immediately. If we are unlucky, it runs until the destroyed part of the program is needed and then fails. This problem is shown in Figure 7-8.



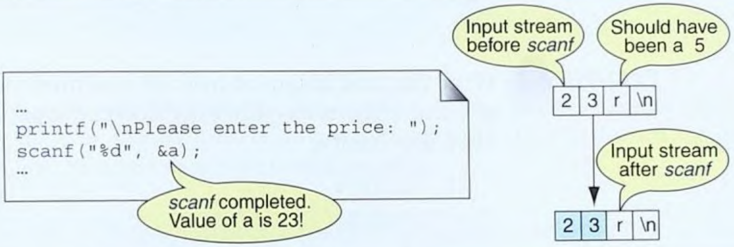
```
...
printf("\nPlease enter number of dependents: ");
scanf ("%4d", a);
...
```

FIGURE 7-8 Missing Address Operator in *scanf*

Data Type Conflict

The second common mistake is a conflict between the format string and the input stream. This occurs, for example, when the format string calls for a numeric value and the input stream contains an alphabetic character. If any conflict occurs between the format string and the input stream, the *scanf* operation aborts. The character that caused the conflict stays in the input stream, and the rest of the input stream (including the troublemaker) remains waiting to be processed. The problem is that the input stream cannot be filled again until it has been completely processed or flushed by the program. The next input operation will read from the unused stream instead of reading from a new input stream. The result: Everything is a mess!

In Figure 7-9, the user meant to enter the number 235 but instead entered 23r. (This is a common error. Note that the *r* is just below and to the left of the 5 on the keyboard.) When *scanf* reads the input stream, it interprets the number 23 but stops with the *r* and returns 1, indicating that one decimal number was successfully formatted. The *r* remains in the input stream, waiting for the next *scanf*. The typical result of this error is a perpetual loop looking for an integer value. Even if we detect and flush the error when the *r* is read, we still have processed the wrong data (23, not 235) in the first read!

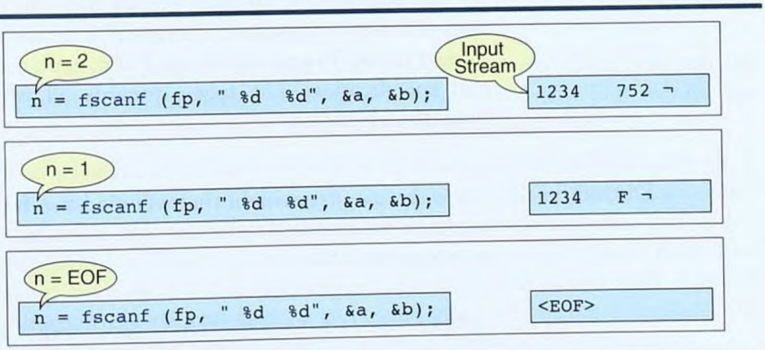


```
...
printf("\nPlease enter the price: ");
scanf("%d", &a);
...
```

Input stream before *scanf*
Should have been a 5
2 3 r \n
Input stream after *scanf*
2 3 r \n
scanf completed. Value of a is 23!

FIGURE 7-9 Data Type Conflict

Figure 7-10 contains three more examples that show the operation of a *fscanf* function. In the first example, the format string requests that two integers be read, and two are successfully read. The value returned by the *scanf* function is 2. In the second example, two more numbers are requested, but only one is available in the input stream. In this case, the value of the *scanf* expression is 1. The third example shows the result if all the data have been read and the input file is at end of file.



```
n = fscanf (fp, " %d  %d", &a, &b);
```

```
n = fscanf (fp, " %d  %d", &a, &b);
```

```
n = fscanf (fp, " %d  %d", &a, &b);
```

FIGURE 7-10 *fscanf* Examples

Input Examples

In this section, we present several examples of *scanf* formatting. Some of them are rarely used, but they illustrate some of the finer points of formatting. We suggest that you cover up the answer until you have tried to write the statement yourself.

EXAMPLE 7-1 Write the *scanf* statement that will read three values—a real number, an integer, and a character— from the standard input file. The input is

```
3.1416 31416 x
```

```
scanf("%f %d %c", &fNum, &iNum, &aChr);
```

EXAMPLE 7-2 Write the *scanf* statement that will read three values—a real number, an integer, and a character—from the standard input file. All data are on different lines. The input is

```
3.1416
31416
x
```

```
scanf("%f %d %c", &fNum, &iNum, &aChr);
```

Note that this statement is the same as the first example. Since the newline is whitespace, it makes no difference if the data are separated by newlines, spaces, or tabs. However, the following code fails when it tries to read the character!

```
scanf("%f%d%c", &fNum, &iNum, &aChr);
```

Do you see the mistake? Recall that the character conversion code (c) does *not* skip whitespace. Therefore, it reads the return character from the integer input. To prevent this problem, code a space before the character format specification. The space discards all whitespace characters before it reads the character.

EXAMPLE 7-3 Write the *scanf* statement that will read two fractions, such as 14/26 and 25/66, from the standard input unit and place the numerators and denominators in integer variables.

```
scanf("%d/%d %d/%d", &n1, &d1, &n2, &d2);
```

EXAMPLE 7-4 Write the *fscanf* statement that will read a date, such as 5-10-1936, formatted with dashes (-) into three different integer fields. The input is to be read from a file opened as `spData`.

```
fscanf(spData, "%d-%d-%d", &month, &day, &year);
```

Note that with this statement, an error results if the user enters 5/10/1997.

EXAMPLE 7-5 Write the *fscanf* statement that will read three integers from a file opened as `spData` and assign the number of successfully formatted variables to the identifier `result`.

```
result = fscanf(spData, "%d %d %d", &i, &j, &k);
```

EXAMPLE 7-6 Given a file with four integers in each line, write a *fscanf* statement to read only the first, second, and fourth integer from a line; that is, discard the third element. Read the data from the file `spData`.

```
result = fscanf(spData, "%d%d%*d%d", &a, &b, &d);
```

EXAMPLE 7-7 Write the *fscanf* statement that will read three integers from a file opened as `spData` and assign the number of successfully formatted variables to the identifier `result`. Use a defined constant for the format string.

```
#define FORMAT_3D "%d %d %d"

result = fscanf(spData, FORMAT_3D, &i, &j, &k);
```

Input Stream Concerns

There are two more points we need to discuss about *scanf* and how it handles the input stream.

1. The input stream is buffered. The operating system does not pass the data in the input stream until we press the return key. This means that there is always a return character at the end of the stream.
2. The *scanf* function leaves the return character in the buffer by default. If we want the buffer to be empty, we must read the return character and discard it. We read it using a character conversion specification with a suppress flag. When the return character remains in the buffer, the next *scanf* discards it for all conversion codes except character and scan set. Alternatively, we can force it to be discarded by placing a space at the beginning of the format string or before a conversion specification.

3. The *scanf* function does not terminate until the format string is exhausted, that is, until all specified operations are complete. Whitespace characters in the format string cause whitespace characters in the buffer to be read and discarded. When a whitespace character occurs at the end of the format string, *scanf* reads and discards the return character in the buffer and then waits for more input. It cannot terminate until it finds a nonwhitespace character.

Let's look at some examples.

EXAMPLE 7-8 In this example, we have just one *scanf* call in our program that reads two integers. In this case, the return character remains in the buffer, but it is not a problem. When the program terminates, the operating system flushes all buffers.

```
scanf ("%d %d", &a, &b);
```

EXAMPLE 7-9 In this example, we explicitly consume the return character by reading and suppressing it.

```
scanf ("%d %d %*c", &a, &b);
```

EXAMPLE 7-10 In this example, we let the return key stay in the buffer and let the second call *discard* it. Note that in this case, it does not matter if the next *scanf* follows immediately or after several lines of code. The first conversion specification (%d) at the beginning of the second *scanf* discards the return character left by the first *scanf*. The return character for the second input stream, however, remains in the buffer.

```
scanf ("%d %d", &a, &b);
scanf ("%d %d", &c, &d);
```

EXAMPLE 7-11 When the first format specification in a *scanf* format string uses a character or edit set conversion code, any return character left in the buffer by a previous *scanf* operation must be manually consumed. This can be problematic because the leftover return character is read first as shown below.

```
// Does not work because %c reads the left-over return
scanf ("%d  %d", &num1, &num2);
scanf ("%c %d" &aChr, &num1);
```

One solution would be to put a whitespace character at the beginning of the second *scanf* to discard the previous return character. The whitespace before %c in the second *scanf* discards the return character.

```
// Preferred Solution
scanf ("%d  %d", &num1, &num2);
scanf (" %c %d", &aChr, &num1);
```

Another solution would be to discard the return character at the end of the previous format string. Note that as the number of *scanf* statements in a program grow, the safer and preferred method is to discard the return character at the beginning of the format string. In the following code, the character conversion specification (%*c) consumes the newline character at the end of the stream.

```
Alternate Solution:
scanf ("%d  %d%*c", &num1, &num2);
scanf ("%c %d", &aChr, &num1);
```

EXAMPLE 7-12 The *scanf* function can hang our program. For example, in the following fragment of code, we expect that the *printf* function be called after we entered two integers. The program, however, hangs. The reason is that the whitespace at the end of the control string tells *scanf* to discard any whitespace it sees in the buffer. Therefore, *scanf* deletes our return character. But, to complete its operation, *scanf* must find a return character. It does not matter how many return keys we enter on the keyboard, *scanf* discards all of them looking for a non-whitespace character.

```
// Note space at end of first format string.
scanf ("%d  %d ", &a, &b);
printf("Data entered: %d %d\n", a, b);
scanf ("%d %d", &c, &d);

Sample run:
10 20 <return>
<return>
```

We can force *scanf* to complete by entering any non-whitespace character. Because *scanf* wants to format only two numbers, any character causes it to complete successfully. The following example demonstrates how we can force *scanf* to complete.

```
// Note space at end of first format string.
scanf ("%d  %d ", &a, &b);
printf("Data entered: %d %d\n", a, b);
scanf ("%d %d", &c, &d);

Sample run:
10 20<return>                   // scanf hangs here
x<return>                       // extra character to force end
Data entered: 10 20
```

Note, however, that the extra character we enter (x) remains in the buffer and will be read next time *scanf* is executed. This extra character is undoubtedly not valid for the next read, which means that our program will produce invalid results.

Discarding the return character is different from consuming it. Discarding can be done by whitespaces in the control string; consuming can only be done by reading the return character with a %c.

Output Formatting (*printf* and *fprintf*)

Two print formatted functions display output in human readable form under the control of a format string that is very similar to the format string in *scanf*. When necessary, internal formats are converted to character formats. In other words, it is the opposite of the scan formatted functions.

These functions have the following format:

```
printf("format string", value list)

fprintf(sp, "format string", value list)
```

One of the first differences to note is that the value list is optional. Whereas you always need a variable when you are reading, in many situations, such as user prompts, you display strings without a value list.

Three examples of *printf* output that might be found in a program are shown below. The first is a greeting message at the start of the program, the second displays the result of a calculation, and the last is a closing message.

```
printf ("\nWelcome to Calculator.\n");
printf ("\nThe answer is %6.2f\n", x);
printf ("Thank you for using Calculator");
```

The *fprintf* function works just like *printf* except that it specifies the file in which the data will be displayed. The file can be the standard output (*stdout*) or standard error (*stderr*) files. For example, to write the three previous lines to a report file, we use the following code:

```
fprintf (spReport, "\nWelcome to Calculator.\n");
fprintf (spReport, "\nThe answer is %6.2f\n", x);
fprintf (spReport, "Thank you for using Calculator");
```

Print Conversion Specifications

In Chapter 2 we introduced some of the conversion codes. Table 7-4 shows the flags, sizes, and conversion code for *printf* family.

Argument Type	Flag	Size Specifier	code
integer	-, +, 0, space	hh (char), h (short), none (int), l (long), ll (long long)	d, i
unsigned integer	-, +, 0, space	hh (char), h (short), none (int), l (long), ll (long long)	u
integer (octal)	-, +, 0, #, space	hh (char), h (short), none (int), l (long), ll (long long)	o
integer (hex)	-, +, 0, #, space	hh (char), h (short), none (int), l (long), ll (long long)	x, X
real	-, +, 0, #, space	none (float), l (double), L (long double)	f
real (scientific)	-, +, 0, #, space	none (float), l (double), L (long double)	e, E
real (scientific)	-, +, 0, #, space	none (float), l (double), L (long double)	g, G
real (hexadecimal)	-, +, 0, #, space	none (float), l (double), L (long double)	a, A
character	-	none (char), l (w-char)	c
string	-	none (char string), l (w-char string)	s
pointer			p
integer (for count)		none (int), h (short), l (long)	n
to print %			%

TABLE 7-4 Flags, Sizes, and Conversion Codes for *printf* Family

Flag

There are four output flags: justification, padding, sign, and alternate form. Table 7-5 shows the output flag options.

Flag Type	Flag Code	Formatting
Justification	none	right justified
	–	left justified

continued

TABLE 7-5 Flag Formatting Options

Flag Type	Flag Code	Formatting
Padding	none	space padding
	0	zero padding
Sign	none	positive value: no sign negative value: –
	+	positive value: + negative value: –
	space	positive value: space negative value: –
Alternate	#	print alternative format for real, hexadecimal, and octal.

TABLE 7-5 Flag Formatting Options *(continued)*

Justification The **justification flag** controls the placement of a value when it is shorter than the specified width. Justification can be left or right. If there is no flag and the defined width is larger than required, the value is right justified. The default is right justification. To left justify a value, the flag is set to minus (-).

Padding The **padding flag** defines the character that fills the unused space when the value is smaller than the print width. It can be a space, the default, or zero. If there is no flag defined for padding, the unused width is filled with spaces; if the flag is `0`, the unused width is filled with zeros. Note that the zero flag is ignored if it is used with left justification because adding zeros after a number changes its value.

Sign Flag The **sign flag** defines the use or absence of a sign in a numeric value. We can specify one of three formats: default formatting, print signed values, or prefix positive values with a leading space. Default formatting inserts a sign only when the value is negative. Positive values are formatted without a sign. When the flag is set to a plus (+), signs are printed for both positive and negative values. If the flag is a space, then positive numbers are printed with a leading space and negative numbers with a minus sign.

Alternate Flag The **alternate flag** (#) is used with the real, engineering, hexadecimal, and octal conversion codes. The alternate flag is discussed with the conversion codes to which it applies.

Width

The meaning of the **width** field is conceptually the same for input and output, but in the opposite direction. For input, the width specifies the maximum width. For output specifications, the width provides the minimum output width. However, if the data are *wider* than the specified width, C *prints all the data*.

Precision

Precision is specified as a period followed by an integer. Precision has meaning only for output fields, and it is an error to use it for input. Precision can control the following:[5]

1. The minimum number of digits for integers. If the number has fewer significant digits than the precision specified, leading zeros will be printed.
2. The number of digits after the decimal point in float.
3. The number of significant digits in `g` and `G` output.

Size

The **size** specification is a modifier for the conversion code. Used in combination with the conversion code, it specifies the type of the associated variable, for example, a *long double* (`Lf`). The size codes with their associated conversion codes are explained in Table 7-4, "Flags, Sizes, and Conversion Codes for *printf* Family."

Conversion Codes

With the exception of string and pointer, this section discusses the input conversion codes. The pointer conversion code is discussed in Chapter 9; the string conversion code is discussed in Chapter 11.

Integer (`d` and `i`) The value is formatted as a signed decimal number. The integer format code (`i`) is identical to the decimal code (`d`) for output.

Unsigned (`u`) The value is formatted as a unsigned decimal number.

Octal and Hexadecimal (`o`, `x`, `X`) The value is formatted in either octal or hexadecimal, as indicated by the conversion code. The alpha hexadecimal codes are printed in lowercase if the `x` code is used and in uppercase for the `X` code. If you are not familiar with octal or hexadecimal numbers, see Appendix D, "Numbering Systems."

When the alternative format flag is used, the hexadecimal values are prefixed with `ox` or `OX` and octal numbers are printed with a zero prefix.

Real (`f`) The real format code (`f`) is discussed in Chapter 2. If the precision is 0, the decimal point is printed only if the alternate flag (*#*) is also present.

Scientific notation (`e`, `E`, `g`, `G`, `a`, `A`) These codes are the same as described for the *scanf*.

The `e` and `E` codes format the number using (`f`), precision may be used to control the number of digits after the decimal point. If no precision is specified, six digits are used. If the precision is 0, no digits are used after the

[5] In Chapter 11, "Strings," we introduce another use of precision.

decimal point and the decimal point itself is shown only when the alternate form flag (`#`) is used.

The `g` and `G` codes are also scientific notation. However, the scientific notation is used only when the exponent is less than –4 or greater than the specified precision. Otherwise, the real formatting rules are used. When the value is formatted in scientific notation, the uppercase `G` formats the number with an uppercase `E`.

The `a` and `A` codes generate signed hexadecimal fractional representations with a decimal power-of-two exponent. The formatted result with `A` is formatted as `±0Xh.hhhP±dd`, where h represents hexadecimal digits and d represents decimal digits.

Count (`n`) To verify the number of output characters, we use the `n` conversion code. This code requires a matching variable into which *printf* places the count of the characters output. Since the operation places the results in the variable, its address must be used in the parameter list. In the following example, `count` is a short integer that is to receive the number of characters written (note the address operator):

```
printf("%d %8.2f %d %hn", i, x, j, &count);
```

Percent (`%`) The percent sign as a conversion code is used to output a percent sign. For example, assume that `grade` contains a student's score expressed as a percentage. The following statement could then be used to print the grade in the format 93.5%. Note that there is no output value for the percent conversion code.

```
printf("%4.1f%%", grade);
```

EXAMPLE 7-13 Demonstrate Scientific Formatting

The code fragment shown below demonstrates the use of scientific formatting. We print the same real number first in real number format (%f), then in three scientific notations (`%e`, `%G`, and `%A`).

```
double x = 1234.5678;
printf("|%#.4f| |%#.4e| |%#.4G| |%#.4A|", x, x, x, x);
Results:
|1234.5678| |1.2346e+03| |1235.| |0X1.34A4P+10|
```

The fourth output needs a little discussion. To verify that it is correct, we must convert the hexadecimal number to decimal. Using the following calculation, we see that 1.34A4 is approximately equal to 1234.56.

$$(\mathbf{1} \times \mathbf{16}^{\mathbf{0}} + \mathbf{3} \times \mathbf{16}^{-\mathbf{1}} + \mathbf{4} \times \mathbf{16}^{-\mathbf{2}} + \mathbf{A} \times \mathbf{16}^{-\mathbf{3}} + \mathbf{4} \times \mathbf{16}^{-\mathbf{4}}) \times \mathbf{2}^{\mathbf{10}}$$

Output Side Effect and Value

The side effect of the print functions is to write text data to the output file. The value it returns is the number of characters written. If an error occurs, EOF, which is defined in the *stdio.h* header file, is returned. These concepts, which are summarized in Figure 7-11, are similar to the side effects and value discussed with scan functions.

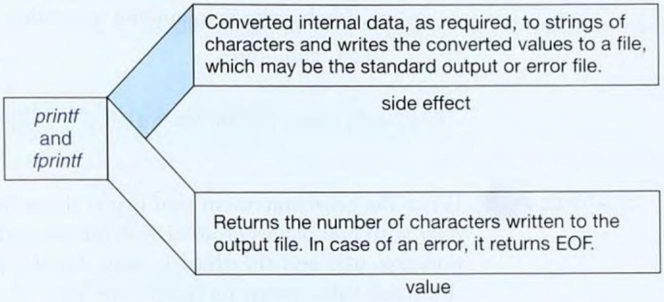

Converted internal data, as required, to strings of characters and writes the converted values to a file, which may be the standard output or error file.
side effect
printf
and
fprintf
Returns the number of characters written to the output file. In case of an error, it returns EOF.
value

FIGURE 7-11 Side Effect and Value of *printf*

Output Examples

In this section we present several examples of print formatting. Although some of them are rarely used, they do illustrate some of the finer points of formatting. The sample output is shown in bold characters. You may want to cover up the answer until you have answered the question yourself.

EXAMPLE 7-14 Write the print statement that prints three columns of numbers. The first column contains a two-digit integer, the second column contains up to seven digits, and the third column contains a *float* with four integral numbers and three decimal places.

```
15    10         15.010
78    1234567    1234.123
```

```
printf("%2d %7ld %8.3f", i, j, x);
```

EXAMPLE 7-15 Write the print statement that prints the tax due as stored in a *float* named x. The output is

```
The tax is:  233.12
```

```
printf("The tax is: %8.2f\n", x);
```

EXAMPLE 7-16 Modify the previous example to add the words "dollars this year" after the numeric value. The output will then be

```
The tax is   233.12 dollars this year.
```

```
printf("The tax is %8.2f dollars this year\n", x);
```

EXAMPLE 7-17 Write the print statement that prints three integer variables to a file opened as `spOut`. Each write is to append a newline character (¬) to the output.

```
100 200 300¬
```

```
fprintf(spOut, "%d %d %d\n", i, j, k);
```

EXAMPLE 7-18 Write the print statement that prints three floating-point values. The first is a *float* with two places of precision; the second is a *double*, printed in scientific notation (`G`); and the third is *long double*, printed using scientific notation (`E`). Each value prints on a separate line.

```
3.14
200
4.408946E+00
```

```
fprintf(spOut, "%.2f\n%G\n%E\n", i, j, k);
```

EXAMPLE 7-19 Write the print statement that prints three integral values. The first is a *short*, printed in octal: the second is an *int*, printed using lowercase hexadecimal digits; and the third is a *long*, printed using uppercase hexadecimal digits. End the line with a newline character.

```
25547 7e44 7864CB
```

```
fprintf(spOut, "%ho %x %lX\n", i, j, k);
```

EXAMPLE 7-20 Write the print statement that prints three real numbers in scientific notation (`e` or `g`) as shown below. Use the alternate format flag so that the value prints with a decimal point even if it is zero.

```
3.1 7.893E-05    0.
```

```
printf("%#5.2g %#9.4G %#5.0g\n", i, j, k);
```

EXAMPLE 7-21 Write the print statement that prints a decimal number in decimal, hexadecimal, and octal format.

```
256 0X100  0400
```

```
printf("%5d %#5X %#5o\n", x, x, x);
```

File Sample Programs

In this section we develop four small programs that demonstrate various aspects of reading and writing files.

Read and Print File

Program 7-3 is a simple program that reads a file of integers and prints them out.

PROGRAM 7-3 Read and Print Text File of Integers

```
/* Read a text file of integers, and print the contents.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
// Local Declarations
   FILE* spIn;
   int   numIn;

// Statements
   spIn = fopen("P07-03.DAT", "r");
   if (spIn != NULL)
      {
       printf("Could not open file\a\n");
       exit  (101);
      } // if open fail

   while ((fscanf(spIn, "%d", &numIn)) == 1)
      printf("%d ", numIn);

   return 0;
}  // main
```

```
Results:
1 2 3 4 5 6 7 8 9 10
```

Program 7-3 Analysis Note how we open the file in this small program. To make it easier to read the code, we split the file open and the test for successful open into two statements. This does not affect the program's efficiency and makes it much easier to code and read the statements.

Now look at the read in the *while* loop. We loop as long as the *fscanf* succeeds, as indicated by a return value of 1. When the end of file is reached, the *while* statement will be false and the program terminates. There is one slight risk to this program: If a read error is caused by a bad disk, we get the same results. In Chapter 13 we show you how to detect this type of error.

Copy File

In Program 7-4, we copy a text file of integers to a new file.

PROGRAM 7-4 Copy Text File of Integers

```
/* Copy a text file of integers.
    Written by:
    Date:
*/
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
// Local Declarations
  FILE* spIn;
  FILE* spOut;
  int   numIn;
  int   closeResult;

// Statements
  printf("Running file copy\n");
  spIn = fopen("P07-03.DAT", "r");
  if (!spIn)
    {
     printf("Could not open input file\a\n");
     exit  (101);
    } // if open fail

  spOut = fopen("P07-04.DAT", "w");
  if (!spOut)
    {
     printf("Could not open output file\a\n");
     exit  (102);
    } // if open fail

  while ((fscanf(spIn, "%d", &numIn)) == 1)
    fprintf(spOut, "%d\n", numIn);

  closeResult = fclose(spOut);
```

continued

PROGRAM 7-4 Copy Text File of Integers *(continued)*

```
36       if (closeResult == EOF)
37          {
38           printf("Could not close output file\a\n");
39           exit  (201);
40          } // if close fail
41 
42       printf("File copy complete\n");
43       return 0;
44    }  // main
```

```
Results:
Running file copy
File copy complete
```

Program 7-4 Analysis Because this program has no display output, we start with a message that says we will copy a file, and we end with a message that says the copy succeeded. Users need to know that the program ran. If we don't print these messages, users will not know if the program ran. In fact, it is a good idea to print start and end messages with every program. This is a common standard in large production systems.

Now study the format string in statement 33 carefully. Note that it requires a newline after the data have been written. It is important to use whitespace between the data in the file or it will not be readable. To verify that the data are correctly written, the file should be read and printed by Program 7-3 or a similar program.

Finally, note that we close the output file at the end of the program. While input files don't need to be closed, you should always close output files and check to make sure that they closed successfully. If the close fails, we print an error message. If it doesn't, we print the successful copy message.

Append File

Now let's append some data to the end of a file. After we wrote Program 7-5, we ran it without a file. Because append creates a file if it doesn't exist, the first run created our file. Then we ran it a second time and verified that the new data were appended to the original file.

PROGRAM 7-5 Append Data to File

```
/* Copy a text file of integers.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
```

continued

PROGRAM 7-5 Append Data to File *(continued)*

```
// Local Declarations
   FILE* spAppnd;
   int   numIn;
   int   closeResult;

// Statements
   printf("This program appends data to a file\n");
   spAppnd = fopen("P07-05.DAT", "a");
   if (!spAppnd)
      {
       printf("Could not open input file\a\n");
       exit  (101);
      } // if open fail

   printf("Please enter first number:  ");
   while ((scanf("%d", &numIn)) != EOF)
      {
       fprintf(spAppnd, "%d\n", numIn);
       printf("Enter next number or <EOF>: ");
      } // while

   closeResult = fclose(spAppnd);
   if (closeResult == EOF)
      {
       printf("Could not close output file\a\n");
       exit  (201);
      } // if close fail

   printf("\nFile append complete\n");
   return 0;
}  // main
```

```
Results:
This program appends data to a file
Please enter first number:  1
Enter next number or <EOF>: 2
Enter next number or <EOF>: 3
Enter next number or <EOF>:^d
File append complete
```

Program 7-5 Analysis This program differs from Program 7-4 in two ways. First, this program has only one disk file. While we could have read the data to be appended from a file, it is more interesting and more flexible to get the data from the keyboard. The second change is the read loop to get the data. Because we are reading from the keyboard, we need to provide prompts for the user.

Student Grades

Programs 7-3 to 7-5 were all written in *main*. As our last example, let's write a program that reads and writes a student grades file to be turned in at the end of the term. The code is shown in Program 7-6.

PROGRAM 7-6 Student Grades

```
/* Create a grades file for transmission to Registrar.
      Written by:
      Date:
*/
#include <stdio.h>

// Function Declarations
int getStu     (FILE* spStu,
                int* stuID,  int* exam1,
                int*  exam2, int* final);
int writeStu   (FILE* spGrades,
                int   stuID, int   avrg,  char grade);
void calcGrade (int   exam1, int   exam2, int  final,
                int*  avrg,  char* grade);

int main (void)
{
// Local Declarations
   FILE* spStu;
   FILE* spGrades;

   int stuID;
   int exam1;
   int exam2;
   int final;
   int avrg;

   char grade;

// Statements
   printf("Begin student grading\n");
   if (!(spStu = fopen ("P07-06.DAT", "r")))
      {
       printf("\aError opening student file\n");
       return 100;
      } // if open input

   if (!(spGrades = fopen ("P07-06Gr.DAT", "w")))
      {
```

continued

PROGRAM 7-6 Student Grades *(continued)*

```
       printf("\aError opening grades file\n");
       return 102;
      } // if open output

   while (getStu
         (spStu, &stuID, &exam1, &exam2, &final))
      {
       calcGrade (exam1, exam2, final, &avrg, &grade);
       writeStu  (spGrades, stuID, avrg, grade);
      } // while

   fclose (spStu);
   fclose (spGrades);

   printf("End student grading\n");
   return 0;
}  // main

/*=======================getStu=======================
   Reads data from student file.
      Pre   spStu is an open file.
            stuID, exam1, exam2, final pass by address
      Post  reads student ID and exam scores
            if data read   --returns 1
            if EOF or error--returns 0
*/
int getStu (FILE* spStu, int* stuID, int* exam1,
            int*  exam2, int* final)
{
// Local Declarations
   int ioResult;

// Statements
   ioResult = fscanf(spStu, "%d%d%d%d", stuID,
                     exam1, exam2, final);
   if (ioResult == EOF)
       return 0;
   else if (ioResult != 4)
      {
       printf("\aError reading data\n");
       return 0;
      } // if
   else
      return 1;
```

continued

PROGRAM 7-6 Student Grades *(continued)*

```
}  // getStu

/* ==================== calcGrade ====================
   Determine student grade based on absolute scale.
      Pre   exam1, exam2, and final contain scores
            avrg and grade are addresses of variables
      Post  Average and grade copied to addresses
*/
void calcGrade (int  exam1, int   exam2, int final,
                int* avrg,  char* grade)
{
// Statements
   *avrg = (exam1 + exam2 + final) / 3;
   if (*avrg >= 90)
      *grade = 'A';
   else if (*avrg >= 80)
      *grade = 'B';
   else if (*avrg >= 70)
      *grade = 'C';
   else if (*avrg >= 60)
      *grade = 'D';
   else
      *grade = 'F';
   return;
}  // calcGrade

/* ==================== writeStu ====================
   Writes student grade data to output file.
      Pre   spGrades is an open file
            stuID, avrg, and grade have values to write
      Post  Data written to file
*/
int writeStu  (FILE* spGrades, int  stuID,
               int   avrg,     char grade)
{
// Statements
   fprintf(spGrades, "%04d %d %c\n",
                     stuID, avrg, grade);

   return 0;
}  // writeStu
```

```
Results:
Begin student grading
End student grading
```

continued

PROGRAM 7-6 Student Grades *(continued)*

```
Input--------
0090 90 90 90
0089 88 90 89
0081 80 82 81
0079 79 79 79
0070 70 70 70
0069 69 69 69
0060 60 60 60
0059 59 59 59

Output----
0090 90 A \n
0089 89 B \n
0081 81 B \n
0079 79 C \n
0070 70 C \n
0069 69 D \n
0060 60 D \n
0059 59 F \n
```

Program 7-6 Analysis

This program has several points to study. Let's start at the top. We open and close the files in *main*. Often, programmers write subfunctions for program initialization and conclusion, but in this case we decided to write them in *main*. The processing loop uses the return value from `getStu` to control the loop (see statement 44). If data are read, then the return value is true (1); if there is an error or all data have been read, the return value is false (0). This design results in a very simple *while* loop.

Within the loop are three calls: one to read the student data (in the *while* limit test), one to calculate the grade, and one to write the grades file. Study the parameters carefully. We have used a combination of data values and addresses. Note especially the file parameters. In the function headers, the file parameters are coded as `FILE*`, which you should recognize as a file address. We do not use an address operator when we pass the files, however, because they are already addresses. (See statements 44 and 48.)

Now study the results. The student ID is written as four digits with leading zeros because it is really a code, not a number. Note how we created the test data to verify that all of the boundaries in our multiway selection work correctly. To make it easy to verify, we made the student ID the same as the average. With these clues to the expected output, it only takes a quick review of the results to confirm that the program ran successfully.

7.5 Character Input/Output Functions

Character input functions read one character at a time from a text stream. **Character output functions** write one character at the time to a text stream. These functions can be divided into two general groups: input/output functions used exclusively with a terminal device and input/output functions that can be used with both terminal devices and text files. They are summarized in Figure 7-12.

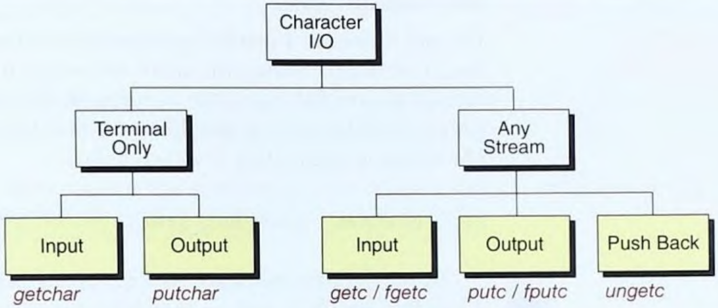


FIGURE 7-12 Character Input/Output Functions

C provides two parallel sets of functions, one for characters and one for wide-characters. They are virtually identical except for the type. Because wide-characters are not commonly used and because their functions operate identically, we limit our discussion to the character type. The wide-character functions are listed in Appendix F.

Terminal Character I/O

C declares a set of character input/output functions that can only be used with the standard streams: standard input (*stdin*), standard output (*stdout*).

Read a Character: *getchar*

The get character function (*getchar*) reads the next character from the standard input stream and returns its value. Only two input conditions stop it: end of file or a read error. If the end of file condition results, or if an error is detected, the functions return EOF. It is our responsibility to determine if some condition other than end of file stopped the reading. The function declaration is shown below.

```
int getchar (void);
```

Note that the return type is an integer and not a character as you might expect. The return type is an integer because EOF is defined as an integer (*int*) in the standard definition *stddef.h* and other header files. There is another reason for this: C guarantees that the EOF flag is not a character. This is true regardless of what character set it is using: ASCII, EBCDIC, or whatever. If you examine the control characters in ASCII, you will find none for end of file. Traditionally, EOF is defined as −1, but this is not prescribed by ANSI/ISO. An individual implementation could therefore choose a different value.

Write a Character: *putchar*

The put character function (*putchar*) writes one character to the monitor. If any error occurs during the write operation, it returns EOF. This may sound somewhat unusual, since EOF is normally thought of as an input file consideration, but in this case it simply means that the character couldn't be written. The function declaration is shown below.

```
int putchar  (int out_char);
```

Again the type is integer. An interesting result occurs with this function: When it succeeds, it returns the character it wrote as an integer!

Terminal and File Character I/O

The terminal character input/output functions are designed for convenience; we don't need to specify the stream. In this section, we describe a more general set of functions that can be used with both the standard streams and a file.

These functions require an argument that specifies the stream associated with a terminal device or a file.

1. When used with a terminal device, the streams are declared and opened by the system—the standard input stream (*stdin*) for the keyboard and the standard output stream (*stdout*) for the monitor.
2. When used with a file, we need to explicitly declare the stream. It is our responsibility to open the stream and associate it with the file.

Read a Character: *getc* and *fgetc*

The get functions read the next character from the file stream, which can be a user-defined stream or *stdin*, and convert it to an integer. If the read detects an end of file, the functions return EOF. EOF is also returned if any error occurs. We discuss how to tell the difference in Chapter 13. The prototype functions are shown below.

```
int     getc   (FILE* spIn);
int     fgetc  (FILE* spIn);
```

While the *getc* and *fgetc* functions operate basically the same, for technical reasons we recommend that only *fgetc* be used. An example of *fgetc* is shown in the following section.

```
nextChar = fgetc (spMyFile);
```

Write a Character: *putc* and *fputc*

The put functions write a character to the file stream specified, which can be a user-defined stream, *stdout*, or *stderr*. For *fputc*, the first parameter is the character to be written and the second parameter is the file. If the character is successfully written, the functions return it. If any error occurs, they return EOF. The prototype functions are shown below.

```
int putc  (int oneChar, FILE* spOut);
int fputc (int oneChar, FILE* spOut);
```

Once again, we recommend that only *fputc* be used. An example is shown below.

```
fputc (oneChar, spMyFile);
```

Note that we have discarded the return value. Although we may occasionally find a use for it, it is almost universally discarded.

Push a Character Back: *ungetc*

The push back functions insert one or more characters into a stream that has been opened for input; that is, it writes to an input stream. When multiple characters are pushed back, they are then read in reverse order; that is the last character pushed is the first character read. While multiple characters can be pushed back, we cannot push back more characters than we read. After a successful push back, when the stream is next read, the character pushed back will be read.

If the operation succeeds, the functions return the character; if they do not succeed, as with too many characters being pushed into the stream, they return EOF.

The *unget* character functions require that the first parameter be a character and that the second parameter be an input stream. The function declarations are shown below.

```
int ungetc (int oneChar, FILE* spData);
```

The intent of these functions is to return one or more characters to the input stream for a subsequent read. This situation might occur when we are reading characters that prompt action, such as in a menu system, and the wrong character is read (see below). In these situations, it is cleaner to reinsert the character into the input stream than to set a flag or otherwise pass the character through several functions.

In the following example, we want to process any option that is a character. Numeric options are processed in another function. Therefore, if we detect that an option is numeric, we put it back on the input stream and exit the function.

```
option = fgetc (stdin);
if (isdigit(option))
    ungetc (option, stdin);
else
    { … }
```

Character Input/Output Examples

This section contains examples of common text file applications.

Create Text File

Program 7-7 reads text from the keyboard and creates a text file. All data are stored in text format with newlines only where input by the user.

PROGRAM 7-7 Create Text File

```
/* This program creates a text file.
      Written by:
      Date:
*/
#include <stdio.h>
int main (void)
{
// Local Declarations
   FILE* spText;
   int   c;
   int   closeStatus;

// Statements
   printf("This program copies input to a file.\n");
   printf("When you are through, enter <EOF>.\n\n");

   if (!(spText = fopen("P07-07.DAT","w")))
      {
       printf("Error opening P07-07.DAT for writing");
       return (1);
      } // if open

   while ((c = getchar()) != EOF)
      fputc(c, spText);

   closeStatus = fclose(spText);
   if (closeStatus == EOF)
      {
```

continued

PROGRAM 7-7 Create Text File *(continued)*

```
29          printf("Error closing file\a\n");
30          return 100;
31         } // if
32
33       printf("\n\nYour file is complete\n");
34       return 0;
35    }  // main
```

```
Results:
This program copies input to a file.
When you are through, enter <EOF>.

Now is the time for all good students
To come to the aid of their college.^d

Your file is complete
```

Program 7-7 Analysis This simple program is the beginning of a text editor. The biggest element missing is *word wrap*. Word wrap prevents a word from being split between two lines on a page.

Study the program carefully and note that we used integer for the character type (c) in statement 10. This is an important point whenever you are checking for end of file, as we did in statement 23. Because EOF will not be correctly represented in a character, we must use integer. But, what about when we have a character? C automatically casts the integer to a character wherever it is necessary.

Copy Text File

Program 7-8 will copy one text file to another text file. It is more generalized than Program 7-4, "Copy Text File of Integers" because it will copy any file data, not just integers.

PROGRAM 7-8 Copy Text File

```
/* This program copies one text file into another.
      Written by:
      Date:
*/
#include <stdio.h>
int main (void)
{
// Local Declarations
   int    c;
   int    closeStatus;
   FILE*  sp1;
   FILE*  sp2;
```

continued

PROGRAM 7-8 Copy Text File *(continued)*

```

// Statements
  printf("Begin file copy\n");

  if (!(sp1 = fopen ("P07-07.DAT", "r")))
    {
     printf("Error opening P07-07.DAT for reading");
     return (1);
    } // if open input
  if (!(sp2 = fopen ("P07-08.DAT", "w")))
    {
     printf("Error opening P07-08.DAT for writing");
     return (2);
    } // if open output

  while ((c = fgetc(sp1)) != EOF)
    fputc(c, sp2);

  fclose(sp1);
  closeStatus = fclose(sp2);
  if (closeStatus == EOF)
    {
     printf("File close error.\a\n");
     return 201;
    } // if close error
  printf("File successfully created\n");
  return 0;
} // main
```

Program 7-8 Analysis This program contains two style points that need comments. First, we have used generic names for the files, sp1 and sp2. Since this program simply copies and creates text files, we cannot give the files names that reflect their data. Better names might be spIn and spOut, but they are also generic.

Second, the program has two potential file open errors. We use different error codes for those operating systems whose job control can distinguish between different completion codes. We have also included the file names in the error messages so that the user knows exactly which file had the problem.

Finally, a subtle point: Note that we have arranged the local declarations in order of increasing complexity. First, we define the character and integer types: then we define the files. It is good practice to group the definitions by type and in order of increasing complexity.

Count Characters and Lines

Program 7-9 counts the number of characters and lines in a program. Lines are designated by a newline. Also note that the program guards against a file that ends without a newline for the last line.

PROGRAM 7-9 Count Characters and Lines

```
/* This program counts characters and lines in a program.
      Written by:
      Date:
*/
#include <stdio.h>
int main (void)
{
// Local Declarations
   int   curCh;
   int   preCh;
   int   countLn = 0;
   int   countCh = 0;
   FILE* spl;

// Statements
   if (!(spl = fopen("P07-07.DAT", "r")))
      {
       printf("Error opening P07-07.DAT for reading");
       return (1);
      } // if open error

   while ((curCh = fgetc(spl)) != EOF)
      {
       if (curCh != '\n')
           countCh++;
       else
           countLn++;
       preCh = curCh;
      } // while

    if (preCh != '\n')
        countLn++;

    printf("Number of characters: %d\n", countCh);
    printf("Number of lines     : %d\n", countLn);
    fclose(spl);
  return 0;
 }  // main
```

```
Results:
Number of characters: 74
Number of lines:       2
```

Program 7-9 Analysis Program 7-9 is rather straightforward. The only real problem is in making sure that the last line is counted even if there is no newline. We can check this at the end of the file by making sure that the last character we read, stored in preCh, was a newline. If it isn't, we add 1 to the line count.

Once again, we have used the integer type for our input character. Everything works correctly, even statement 31, which compares the integer type (preCh) to a character literal.

Count Words In File

Program 7-10 counts the number of words in a file. A word is defined as one or more characters separated by one or more whitespace characters—that is, by a space, a tab, or a newline.

PROGRAM 7-10 Count Words

```
/* Count number of words in file. Words are separated by
   whitespace characters: space, tab, and newline.
      Written by:
      Date:
*/
#include <stdio.h>

#define WHT_SPC\
         (cur == ' ' || cur == '\n' || cur == '\t')
int main (void)
{
//Local Declarations
   int   cur;
   int   countWd = 0;
   char  word = 'O';       // O out of word: I in word
   FILE* spl;

// Statements
   if (!(spl = fopen("P07-07.DAT", "r")))
      {
       printf("Error opening P07-07.DAT for reading");
       return (1);
      } // if file open error
   while ((cur = fgetc(spl)) != EOF)
      {
       if (WHT_SPC)
           word = 'O';
       else
           if (word == 'O')
              {
               countWd++;
               word = 'I';
```

continued

PROGRAM 7-10 Count Words *(continued)*

```
              } // else
      } // while
   printf("The number of words is: %d\n", countWd);

   fclose(spl);
   return 0;
}  // main
```

```
Results:
The number of words is: 15
```

Program 7-10 Analysis The selection logic for this problem is similar to the previous program. We must determine when we are in a word and when we are between words. We are between words when we start the program and whenever we are at whitespace. To keep track of where we are, we use a flag, `word`. When `word` contains the letter `I`, we are in a word; when it contains the letter `O`, we are out of a word. We increment the counter only at the beginning of a word, when we set the word flag to `I`.

Note that the problem handles multiple whitespace characters by simply setting the word flag to `O`. Note also how we use a preprocessor *define* statement to define whitespace. This has no effect on the efficiency of the program, but it makes it easier to read.

7.6 Software Engineering

Testing Files

Testing files can be a very difficult task for two reasons. First, many errors cannot be created through normal testing. Among these types of errors are those created by bad physical media, such as disk and tapes. Often, disks and tapes become unreadable. Good programs test for these types of errors, but they are difficult to create.

Second, there are so many ways things can go wrong. The test plan therefore needs to ensure that all possibilities have been tested. Chief among them is the case where a user enters a non-numeric character while trying to enter a number. If we don't provide for this situation, our program will not work properly. In fact, in some situations, it will not work at all.

Testing for Valid User Input

Consider the following code to read an integer value. If the user miskeys the data, the program will go into an infinite loop.

```
printf("\nPlease enter Number of Units Sold: ");
while (scanf("%d", &unitsSold) != 1) \
      // scanf returns 1 if number read correctly
      printf("\nInvalid number. Please re-enter.\n");
```

In this case, it looks as if the programmer has done everything correctly. The program checks for valid data being returned. A user prompt shows what data should be entered; the program checks for valid data being read; and if there were errors, provides the user with a good error message and repeats the *scanf*. What is wrong?

The problem lies in what *scanf* does when it finds an invalid first digit. If the number is entirely wrong, it leaves the data pending in the input stream. This means that while the user sees the error message, the invalid data are still in the input stream waiting for a *scanf* that will properly read and process them. Remember that *scanf* thinks that these "invalid data" are the beginning of the next field. It is our job as a programmer to get rid of the "bad data." We can do this with a small piece of code that is commonly named `FLUSH`. Its purpose is to read through the input stream looking for the end of a line. When it finds the end of the line, it terminates. Let's first look at the code, then we will look at an easy way to implement it.

```
while (getchar( ) != '\n')
   ;
```

Examine this statement carefully. All it does is get a character and then throw it away. That is, the character it reads is not saved. Now examine the expression that is controlling the *while*. If *getchar* reads any character other than a newline, the statement is true. If it reads a newline, then the statement is false. The *while* statement will therefore read the input stream and throw away all characters until it sees a newline. When it finds the newline, it will throw it away too, but at that point, the loop stops. In other words, it flushes the input stream to a newline character. This is exactly what we want to do when the user accidentally keys the wrong character when the program expects a digit.

One more point. Note that we coded the null statement on its own line. This is to make it clear that we intended to code a null statement. It also eliminates a warning message from the compiler—good compilers look for common errors like unintended semicolons.

Now, what's the easiest way to implement this handy statement? Well, we could simply code it everywhere you need it, but chances are that we would make some mistakes that would have to be found and debugged. A better solution is to use the preprocessor *define* declarative and code the statement only once, as shown below.

```
#define FLUSH while (getchar( ) != '\n')
```

Note that there is no semicolon at the end of the *define* declarative. We could have put one there, but it's better to put the semicolon after the FLUSH in the program. Our error code can now be changed to handle this type of error, as shown in Program 7-11.

PROGRAM 7-11 Handling Errors—the Right Way

```
#define FLUSH while (getchar() != '\n')
...
   printf("\nPlease enter Number of Units Sold: ");
   while (scanf("%d", &unitsSold) != 1)
      {
       // scanf returns 1 if number read corrrectly
       FLUSH;
       printf("\aInvalid number. Please re-enter: ");
      } // while
```

Value Errors

In Chapter 6, we discussed some of the techniques for data validation. But the subtleties of data validation can be quite complex. Consider another type of human error, the partially correct input. In our program above, we assumed that the user entered invalid data for the first digit of the number. What if the error occurs on the second or third digit? Then the *scanf* function is happy, for the time being anyway, and returns a 1, indicating that it read one number successfully. How do we guard against this?

The best way is to echo the input to the user and ask for verification that it is correct. Although this greatly slows down the input process, for critical data it is necessary. The code for this situation is shown in Program 7-12. Here we present it as a complete function.

PROGRAM 7-12 Handling Errors with Explanations

```
/* This function reads the units sold from the keyboard
   and verifies the data with the user.
      Pre   nothing
      Post  units Sold read, verified, and returned
*/
int getUnitsSold (void)
{
// Local Declarations
   int  unitsSold;
   bool valid;

// Statements
do
   {
    printf("\nPlease enter Number of Units Sold: ");
    while (scanf("%d", &unitsSold) != 1)
          {
           FLUSH;
          printf("\aInvalid number. Please re-enter: ");
          }  // while
      printf("\nVerify Units Sold: %d: ", unitsSold);
      printf("<Y> correct: <N> not correct: \n");
      FLUSH;
      if (toupper(getchar ()) ==  'Y')
          valid = true;
      else
          {
           FLUSH;
           printf("\nYou responded 'no.' ");
           printf("Please re-enter Units Sold\n");
           valid = false;
           } // if
   } while (!valid);
 return unitsSold;
 } // getUnitsSold
```

Program 7-12 Analysis This function merits some discussion. First, note the good user communication throughout the function. It begins with a complete prompt and provides clear error messages whenever problems are detected.

We have implemented this logic with a *do...while* statement, which always loops at least once, since we know that there will always be input. This is the standard loop for validating user input. Within the loop are two different validation steps. The first tests for totally invalid input, the second asks the user to verify the input. If either test indicates a problem, the input is flushed. Note that the user messages are different depending on the circumstances.

The function cannot end unless the `valid` flag is true. The *if* statement in the loop will set it true if the user replies positively. Otherwise, it is set false and the input is flushed. This code again demonstrates two principles of good human engineering. The *if* statement is coded in a positive manner, and the expected event, good input, is checked first.

Many other things can go wrong when we are reading a file, but these two examples cover most of them.

Data Terminology

Computer specialists who deal with data use a set of specific terms to describe their data. These terms deal with data that are stored in files. What we call a variable in our program, they call a **field** or a **data item**. A field is the smallest named unit of data in a file. If we were working with data about the ten western states in the continental United States, we would have data like Name, Capital, Number, Square Miles, Population, and Number of Counties. The first two fields are strings (delimited arrays of characters that we study in Chapter 11) and the last four are integers.

These six fields grouped together make up a state record. A **record** is a collection of related data, in this case state data, treated as a unit. Each record has a **key**, one or more fields that uniquely identify the record. In our state's record, the key could be the name. Names normally do not make good keys, because they are not guaranteed to be unique. A better choice is the state number, which represents the order in which the states entered the union. This field is guaranteed to be unique.

With text files we cannot create a record. We must wait for binary files and structures to do that. But we can simulate a record by grouping these data on the same line, with each field separated from the next by whitespace. The data for the ten western states are shown in Table 7-6.

State	Capital	No.	Sq. Miles	Population	No. Counties
Arizona	Phoenix	48	113,508	3,447,100	15
California	Sacramento	31	156,299	33,871,648	58
Colorado	Denver	38	103,595	4,301,251	63
Idaho	Boise	43	82,412	1,293,953	44
Montana	Helena	41	145,388	902,195	56
Nevada	Carson City	36	109,894	1,998,257	16

continued

TABLE 7-6 Ten Western States (2000 Census)

State	Capital	No.	Sq. Miles	Population	No. Counties
New Mexico	Santa Fe	47	121,335	1,819,046	33
Oregon	Salem	33	96,184	3,421,399	36
Washington	Olympia	42	66,511	5,894,121	39
Wyoming	Cheyenne	44	96,989	493,782	23

TABLE 7-6 Ten Western States (2000 Census) *(continued)*

Data can be logically organized to provide more meaning. Although computer scientists normally store this type of data using binary files, there is no reason it can't be stored in a text file as well. We will return to the discussion again when we study strings in Chapter 11 and binary files in Chapter 13.

7.7 Tips and Common Programming Errors

1. To print a percent sign, you need to use two tokens (`%%`).
2. After you have finished working with a file, you should close it.
3. If you open a file for writing, a new file will be created. This means that if you already have an existing file with the same name, it could be deleted.
4. If you want to write at the end of an existing text file, open the file for appending, not for writing.
5. C output is right justified when using *printf* and *fprintf* functions. If you want your data to be left justified, use the left-justify flag (–) explicitly.
6. It is a compile error to misplace the file pointer in the parameter list. The file pointer is coded last in all file functions except *fscanf* and *fprintf*, in which it is coded first.
7. An error that may cause your program run to terminate is to use a format code that does not match the variable type being read.
8. An error that will give invalid output is to use a format code that does not match the variable being printed.
9. Several common errors are created when a file operation is included in a selection statement. Three are shown below.

   a. The following code does not properly test to make sure that the file was opened correctly:

   ```
   if (sp = fopen() != NULL)          // logic error
   if ((sp = fopen()) != NULL)        // good code
   ```

   b. The following code does not store the character read in `ch`. It stores the logical result of the compare in `ch`.

   ```
   if (ch = getchar() != '\n')        // logic error
   if ((ch = getchar()) != '\n')      // good code
   ```

   c. In a similar way, the following code is invalid:

   ```
   // logic error
   while (ioResult = scanf(…) != EOF)

   // good code
   while ((ioResult = scanf(…)) != EOF)
   ```

   d. It is a logic error to define `ioResult` in the above statement as a character because `EOF` (–1) cannot be stored in a character.

10. It is a logic error to code a whitespace character at the end of a *scanf* statement.

11. It is a run-time error to attempt to open a file that is already opened.

12. It is a run-time error to attempt to read a file opened in the write mode or to write a file opened in the read mode.

7.8 Key Terms

alternate flag
append mode
auxiliary storage devices
binary file
buffer
character input
output functions
close function
data item
field
field width
file
file mode
file type
filename
hexadecimal conversion
key
leading zero padding
open function
precision
read mode
record
right justification
secondary storage device
signed
size
space flag
standard error
standard file stream
text file
value error
write mode

7.9 Summary

- A file is a collection of related data treated as a unit.
- Data in a text file are stored as human-readable characters. The data in a text file are usually divided into lines separated by a newline character.
- A stream is a sequence of elements in time.
- A stream is associated with a file in C.
- There are three standard file streams in C: standard input, standard output, and standard error.
- The standard input is associated with the keyboard. The standard output and the standard error are associated with the monitor.
- The standard file streams can be accessed respectively using *stdin, stdout,* and *stderr*. These are pointers (addresses) to tables (structures) containing the information about the standard streams.
- To manipulate and access files, there are different types of input/output functions.

- ❑ The open/close functions, *fopen* and *fclose*, are used to open and close the association between external files and internal streams.
- ❑ A file in C can be any of the three basic modes: reading, writing, and appending.
- ❑ When a file is opened in reading mode, the file marker is positioned at the beginning of the existing file. The file is ready to be read.
- ❑ When a file is opened for writing, the file marker is positioned at the beginning of a newly created empty file, before the end-of-file character. The file is ready to be written.
- ❑ When a file is opened for appending, the marker is positioned at the end of the existing file (or at the beginning of a newly created file), before the end-of-file marker. The file is then ready to be written.
- ❑ Formatted input/output functions allow us to read data from and write data to files character by character while formatting them to the desired data types such as *char*, *int*, and *float*. The functions *scanf* and *fscanf* are used for reading. The functions *printf* and *fprintf* are used for writing.
- ❑ Character input/output functions allow us to read or write files character by character. The functions *fgetc*, *getc*, and *getchar* can be used for reading. The functions *fputc*, *putc*, and *putchar* can be used for writing.
- ❑ C99 added a set of wide character functions. They are *fgetwc*, *getwc*, *getwchar*, *fputwc*, *putwc*, and *putwchar*.
- ❑ When we are reading data from files, we should validate the data.

7.10 Practice Sets

Review Questions

1. User files are created using the `FILE` type.

   a. True
   b. False

2. If a file does not exist, the append mode returns an error.

   a. True
   b. False

3. The conversion specification starts with an asterisk (*) and ends with the conversion code.

   a. True
   b. False

4. The space flag is used to left-justify output in the *printf* function.

   a. True
   b. False

5. One of the problems with testing files is that some errors, such as physically bad media, cannot be created so that they can be tested.

   a. True
   b. False

6. Which of the following is considered auxiliary storage?

   a. disk
   b. random access memory (RAM)
   c. read only memory (ROM)
   d. tape
   e. both disk and tape

7. Which of the following is not a standard file stream?

   a. *stdin*
   b. *stderr*
   c. *stdfile*
   d. *stdout*

8. The C library that contains the prototype statements for file operations is

   a. *file.h*
   b. *proto.h*
   c. *stdfile.h*
   d. *stdio.h*
   e. *stdlib.h*

9. If a file is opened in the `r` mode, C

   a. opens the file for reading and sets the file marker at the beginning.
   b. opens the file for reading and sets the file marker at the end.
   c. opens the file for writing and sets the file marker at the beginning.
   d. opens the file for writing and sets the file marker at the end.
   e. returns a null file address indicating an error if the file exists.

10. If a file is not opened successfully, C

    a. aborts the program.
    b. continues processing with the next statement after the open.
    c. changes the file mode to write and opens a new file.
    d. displays a message and waits for an answer.
    e. returns a file address.

11. The ______________ specifies the type of data that are being formatted in formatted input/output.

    a. conversion code
    b. data code
    c. field code
    d. format code
    e. pcode

12. The two input functions that read text data and convert the data to the types specified by a format string are

    a. *fread* and *readd*. *read* and *readf*
    b. *fscanf* and *fprintfe*. *scanf* and *printf*
    c. *fscanf* and *scanf*

13. Which of the following statements about the formatted input function (*scanf*) is false?

    a. *scanf* implicitly casts data to match the conversion code and its corresponding address parameter.
    b. *scanf* reads data from the keyboard and formats it according to conversion specifications.
    c. *scanf* returns an integer containing the number of variables successfully formatted for `EOF`.
    d. *scanf* stops formatting data when an inappropriate character for the type being formatted is encountered.
    e. To function properly, *scanf* requires at least one address.

14. The ________________ function reads the next character from the standard input stream and returns its value.

    a. *fgetc*
    b. *getc*
    c. *getchar*
    d. *readchar*
    e. *ungetc*

15. When *scanf* finds invalid data in the input stream, it

    a. flushes the input to a newline.
    b. prints an error message and terminates the program.
    c. prints an error message and terminates the *scanf* function.
    d. skips the data and formats the next conversion specification, if any.
    e. terminates the *scanf* function and leaves the invalid character in the input stream.

Exercises

16. Given the following declaration:

```
int   i1;
int   i2;
float f1;
char  c1;
char  c2;
char  c3;
```

and the following line of data

```
14  23  76  CD
```

what would be the value of i1, i2, f1, c1, c2, and c3 after the following statement?

```
scanf("%d %d %f %c %c %c",
      &i1, &i2, &f1, &c1, &c2, &c3);
```

17. Given the following declaration:

```
int   i1;
int   i2;
float f1;
char  c1;
char  c2;
```

and the following line of data

```
14.2  67  67.9
```

what would be the value of i1, i2, f1, c1, and c2 after the following statement?

```
scanf("%d %c %c %i  %f ",
      &i1, &c1, &c2, &i2, &f1);
```

18. Given the following declaration:

```
int   i1;
int   i2;
int   i3;
char  c1;
char  c2;
char  c3;
```

and the following line of data:

```
c145d123 34.7
```

what would be the value of i1, i2, i3, c1, c2, and c3 after the following statement?

```
scanf("%c%c%d%c%d%d",
      &c1, &c2, &i1, &c3, &i2, &i3);
```

19. What would be printed from the following program segment?

```
int   i1 = 123;
int   i2 = -234;
int   i3 = -7;
float f1 = 23.5;
float f2 = 12.09;
float f3 = 98.34;
char  c1 = 65;
char  c2 = '\n';
char  c3 = 'E';

printf("%06d, %06d, %06d %c",
       i1, i2, i3, c2);
printf("%-6d, %%, \", \\t, %-06d", i2, i1);
printf("%c %d ", c1, c2);
printf("%c %c %#x", c1 + 32, c3 + 3, c2 + 5);
```

20. What output is produced from the following program? Include in your answer the contents of any files created.

```
#include <stdio.h>
int main (void)
{
 FILE* sp;
 int   i;
 char  ch;
 sp = fopen("TEST.DAT", "w");
 for (i = 2; i < 20; i += 2)
     fprintf(sp, "%3d", i);
 fclose(sp);
 sp = fopen("TEST.DAT", "r");
 while ((ch = fgetc(sp)) != EOF)
     if (ch == ' ')
        fputc('*', stdout);
     else
        fputc(ch, stdout);
 return 0;
} // main
```

Problems

For the problems and projects in this chapter, you need to create a test file using your text editor. End each line with a newline character; do not write in paragraph format. We suggest the Gettysburg Address as a suitable subject for these files:

> Four score and seven years ago our fathers brought forth on this continent
> a new nation, conceived in liberty, and dedicated to the proposition that all

continued

men are created equal. Now we are engaged in a great civil war, testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war. We have come to dedicate a portion of that field as a final resting-place for those who here gave their lives that that nation might live. It is altogether fitting and proper that we should do this. But in a larger sense, we cannot dedicate, we cannot consecrate, we cannot hallow, this ground. The brave men, living and dead, who struggled here have consecrated it far above our poor power to add or detract. The world will little note, nor long remember what we say here, but it can never forget what they did here. It is for us the living, rather to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us, to be here dedicated to the great task remaining before us, that from these honored dead we take increased devotion to that cause for which they gave the last full measure of devotion; that we here highly resolve that these dead shall not have died in vain; that this nation, under God, shall have a new birth of freedom, and that government of the people, by the people, for the people, shall not perish from the earth.

21. Write a function that appends one file at the end of the other.

22. Write a function that appends a file to itself.

23. Write a function that accepts a file of varying-length lines and changes it to a formatted file with 60 characters in each line.

24. Write a function that calculates the average number of characters per line in a file.

25. Write a function that deletes the last line of any file.

26. Write a function that deletes the blank lines in a file. A blank line is a line with only one single character in it: newline.

27. Write a program that prints itself.

28. Write a program that copies one text file to another and inserts blank lines between paragraphs in the new file. Paragraphs are identified by a newline character.

29. Write a program to copy only lines beginning with a user-specified character.

30. Write a program to parse words onto separate lines; that is, locate and write each word to its own line. Words are defined as one or more characters separated by whitespace.

31. When *scanf* encounters an error, the invalid data are left in the input stream, sometimes making it impossible to continue. Write a function that reads three pieces of numeric data. If an error is detected (return value not `EOF` but less than 3), flush the erroneous data. (See "Software Engineering" for details on how to write a flush statement.) Then test the

function by entering numeric data and alphabetic data in different sequences.

32. Write a program to insert a blank line after the seventh line in a file.

33. Write a program to delete the sixth line in a file. Do not change the sixth line to a blank line; delete it completely.

34. Write a program to insert a blank line after each line in a file. In other words, double-space the text.

35. Write a program to duplicate the fourth line in a file.

36. Write a program to copy a file, deleting the first two characters of each line. (Do not replace the characters with blanks.)

37. Write a program to copy a file, inserting two space characters at the beginning of each line. In other words, each line will be shifted two characters to the right.

38. Write a program that copies the 21st character of each line in a file to a new file. All extracted characters are to be on the same line. If a line in the input file has fewer than 21 characters, write the last character. If a line is blank—that is, if it consists of only whitespace—then copy nothing. At the end of file, write a newline to the new file and close it.

39. Write a program that writes the odd-numbers between 300 and 500 to a text file.

40. Write a function that writes the multiples of `num` between `lowLimit` and `highLimit` to a text file. Assume that `lowLimit` < `highLimit` and that `num` is not equal to zero. Then write a test driver to validate your function.

41. Write a program to read a set of scores from a text file, count the scores over 90, copy the scores over 90 to a new file, and print the number of scores over 90 to the monitor.

42. Modify Problem 41 to count and create a file for the ranges 0 to 30, 31 to 60, 61 to 90, and over 90. Print the statistics for each range at the end of the program.

Projects

43. Write a program that will read a text file and count the number of alphabetic characters (*isalpha*), digits (*isdigit*), punctuation characters (*ispunct*), and whitespace characters (*isspace*) in the file. At the end of the file, the program is to display an appropriate report. (The classifying functions are covered in Chapter 5.)

44. Write a text analyzer program that reads any text file. The program prints a menu that gives the user the options of counting lines, words, characters, or sentences (one or more words ending in a period), or all of the above. Provide a separate function for each option. At the end of the analysis, write an appropriate report.

45. Write a menu-driven text utility program. The program will have the following capabilities:

a. Copy a user-named file to a new file.

b. Append a user-named file to another user-named file.

c. Change the file format to be double-spaced.

d. Remove all blank lines (change a double-spaced file to single-spaced).

e. Display the contents of the file as a series of 60 character lines with no words split between lines.

46. Using an editor, create an inventory file using the data shown in Table 7-7 (do not include the column captions, just the data).

Part No.	Price	Quantity On Hand	Reorder Point	Minimum Order
0123	1.23	23	20	20
0234	2.34	34	50	25
3456	34.56	56	50	10
4567	45.67	7	10	5
5678	6.78	75	75	25

TABLE 7-7 Inventory file data for Project 46

Write a program to read the inventory file and create an inventory report. The report will contain the part number, price, quantity on hand, reorder point, minimum order, and order amount. The order amount is calculated when the quantity on hand falls below the reorder point. It is calculated as the sum of the reorder point and the minimum order less the quantity on hand. Provide a report heading, such as "Inventory Report," captions for each column, and an "End of Report" message at the end of the report. Print the part number with leading zeros.

47. Using an editor, create the employee file shown in Table 7-8.

Employee No.	Department	Pay Rate	Exempt	Hours Worked
0101	41	8.11	Y	49
0722	32	7.22	N	40
1273	23	5.43	Y	39
2584	14	6.74	N	45

TABLE 7-8 Employee File Data for Project 47

Write a program to read the employee file and create a payroll register. The register will contain the following data:

a. Employee number (print left-justified)
b. Department
c. Pay rate
d. Exempt
e. Hours worked
f. Base pay (pay rate * hours worked)

Chapter 8

Arrays

In Chapter 4 we discussed the first derived data type, the function. In this chapter we discuss the second derived type, the array. Figure 8-1 recaps the six derived types and the chapters in which they are covered.

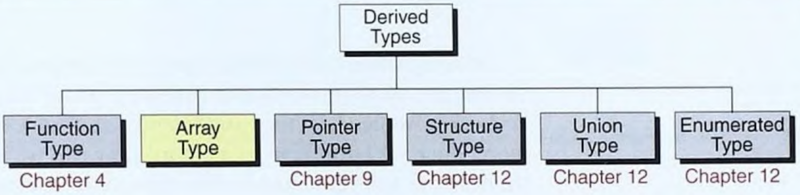


FIGURE 8-1 Derived Types

Objectives

- ❑ To understand the basic concepts and uses of arrays
- ❑ To be able to define C arrays
- ❑ To be able to pass arrays and array elements to functions
- ❑ To understand the classical approaches to sorting arrays: selection, bubble, and insertion sorting
- ❑ To write programs that sort data using the three classical algorithms
- ❑ To be able to analyze the efficiency of a sort algorithm
- ❑ To understand the two classical search algorithms: sequential and binary
- ❑ To write programs that search arrays
- ❑ To be able to design test cases for sorting and searching algorithms
- ❑ To be able to analyze the efficiency of searching algorithms

8.1 Concepts

Imagine we have a problem that requires us to read, process, and print 10 integers. We must also keep the integers in memory for the duration of the program. To begin, we can declare and define 10 variables, each with a different name, as shown in Figure 8-2.

score0
score1
score2
score3
score4
score5
score6
score7
score8
score9

FIGURE 8-2 Ten Variables

Having 10 different names, however, creates a problem: How can we read 10 integers from the keyboard and store them? To read 10 integers from the keyboard, we need 10 read statements, each to a different variable. Furthermore, once we have them in memory, how can we print them? To print them, we need 10 write statements. The flowchart in Figure 8-3 shows a design for reading, processing, and printing these 10 integers.

Although this approach may be acceptable for 10 variables, it is definitely not acceptable for 100 or 1,000 or 10,000. To process large amounts of data we need a powerful data structure, the array. An **array** is a collection of elements of the same data type.

Since an array is a sequenced collection, we can refer to the elements in the array as the first element, the second element, and so forth until we get to the last element. Thus, when we put the 10 integers of our problem into an array, the address of the first element is 0 as shown below.

$\texttt{scores}_0$

In a similar fashion, we refer to the second score as scores_1 and the third score as scores_2. Continuing the series, the last score would be scores_9. We can generalize this concept in the following fashion where the **subscripts** indicate the ordinal number of the element counting from the beginning of the array:

$\texttt{scores}_0, \texttt{scores}_1, \ldots, \texttt{scores}_{n-1}$

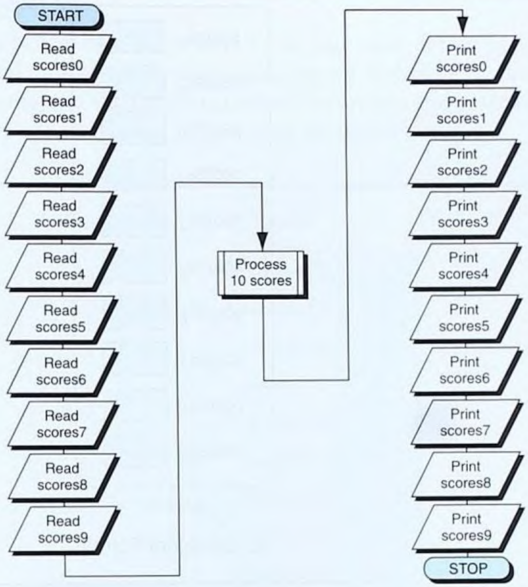


FIGURE 8-3 Process 10 variables

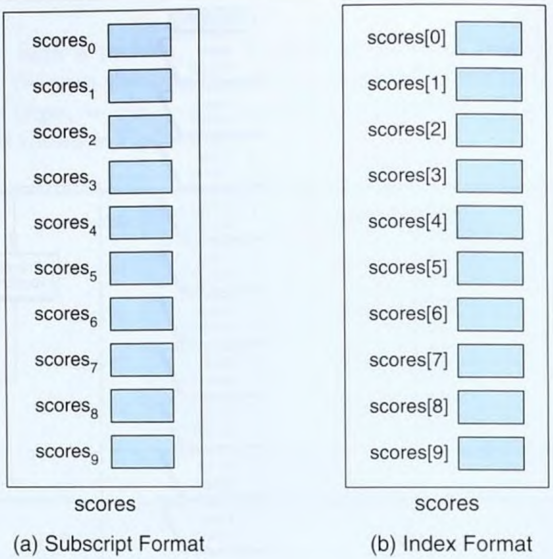
What we have seen is that the elements of the array are individually addressed through their subscripts, a concept shown graphically in Figure 8-4. The array as a whole has a name, `scores`, but each member can be accessed individually using its subscript.

The advantages of the array would be limited if we didn't also have programming constructs that would allow us to process the data more conveniently. Fortunately, there is a powerful set of programming constructs—loops—that makes array processing easy.

We can use loops to read and write the elements in an array; to add, subtract, multiply, and divide the elements; and even for more complex processing such as calculating averages, searching, or sorting. Now it does not matter if there are 1, 10, 100, 1,000, or 10,000 elements to be processed, because loops make it easy to handle them all.

One question remains: How can we write an instruction so that one time it refers to the first element of an array and the next time it refers to another element? The answer is really quite simple: We simply borrow from the subscript concept we have been using. Rather than using subscripts, however, we will place the subscript value in **brackets**. This format is known as *indexing*. Using the index notation, we would refer to scores_0 as

```
scores[0]
```



(a) Subscript Format

(b) Index Format

FIGURE 8-4 An Array of Scores

Following the convention, scores_0 becomes scores[0] and scores_9 becomes scores[9]. Using a typical reference, we now refer to our array using the variable *i*.

```
scores[i]
```

The flowchart showing the loop used to process our 10 scores using an array is seen in Figure 8-5.

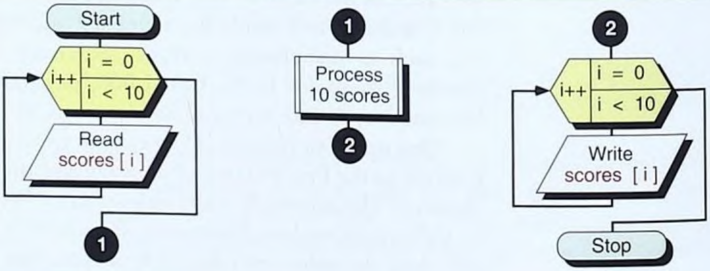


FIGURE 8-5 Loop for 10 Scores

8.2 Using Arrays in C

In this section, we first show how to declare and define arrays. Then we present several typical applications using arrays including reading values into arrays, accessing and exchanging elements in arrays, and printing arrays. Figure 8-6 shows a typical array, named `scores`, and its values.

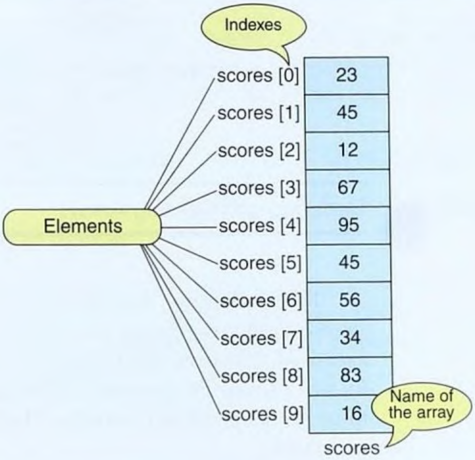


FIGURE 8-6 The Scores Array

C provides for two different array types, fixed-length array and variable-length array. In a **fixed-length array**, the size of the array is known when the program is written. In a **variable-length array**, inroduced in C99, the size of the array is not known until the program is run.

Declaration and Definition

An array must be declared and defined before it can be used. Array declaration and definition tell the compiler the name of the array, the type of each element, and the size or number of elements in the array. In a fixed-length array, the size of the array is a constant and must have a value at compilation time. The declaration format is shown in the following example.

```
type arrayName [arraySize]
```

Figure 8-7 shows three different fixed-length array declarations: one for integers, one for characters, and one for floating-point numbers.

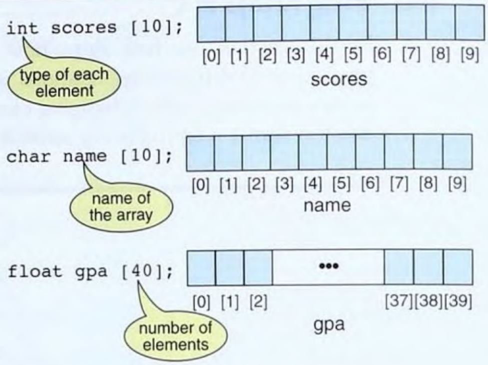


FIGURE 8-7 Declaring and Defining Arrays

The declaration format for a variable-length array is the same as for a fixed-length array except that the array size is a variable. When the program is executed, the array size is determined and the array is defined. Once defined, its size cannot be changed. Following standard C syntax rules, the array size must be declared and initialized before it is used in the variable-length array definition.

```
float salessAry [arySize];
```

Accessing Elements in Arrays

C uses an **index** to access individual elements in an array. The index must be an integral value or an expression that evaluates to an integral value. The simplest form for accessing an element is a numeric constant. For example, given the scores array in Figure 8-6, we could access the first element as shown in the next example:

```
scores[0]
```

The index is an expression, typically the value of a variable. To process all the elements in scores, a loop similar to the following code is used:

```
for (i = 0; i < 10; i++)
    process (scores[i]);
```

You might be wondering how C knows where an individual element is located in memory. In scores, for example, there are nine elements. How does it find just one? The answer is simple. The array's name is a symbolic reference for the address to the first byte of the array. Whenever we use the array's name, therefore, we are actually referring to the first byte of the array. The index represents an offset from the beginning of the array to the element being referenced. With these two pieces of data, C can calculate the address of any element in the array using the following simple formula:

```
element address = array address
                + (sizeof (element) * index)
```

For example, assume that scores is stored in memory at location 10,000. If scores is an integer of type *int*, the size of one element is the size of *int*. Assuming the size of an *int* is 4, the address of the element at index 3 is

```
element address = 10,000 + 4 * 3 = 10,012
```

Storing Values in Arrays

Declaration and definition only reserve space for the elements in the array. No values are stored. If we want to store values in the array, we must either initialize the elements, read values from the keyboard, or assign values to each individual element.

Initialization

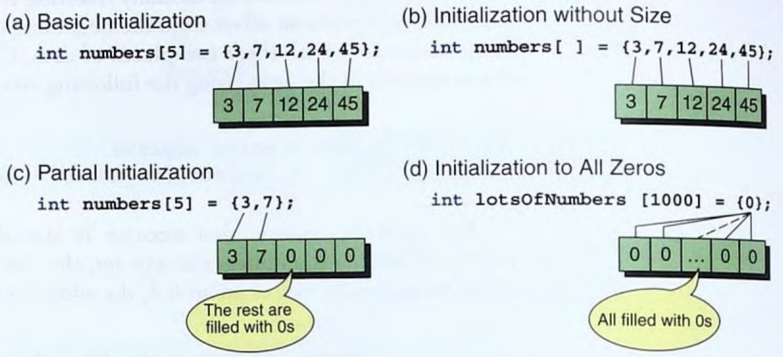
Just as with variables, initialization of the elements in a fixed-length array can be done when it is defined. Variable-length arrays cannot be initialized when they are defined.

For each element in the array, we provide a value. The only difference is that the values must be enclosed in braces and, if there is more than one, separated by commas. It is a compile error to specify more values than there are elements in the array.

Figure 8-8 contains four examples of array initialization. The first example (Figure 8-8(a)) is a simple array declaration of five integers and is typical of the way array initialization is coded. When the array is completely initialized, the programmer does not need to specify the size of the array. This case is seen in Figure 8-8(b). It is a good idea, however, to define the size explicitly, because it allows the compiler to do some checking for errors and is also good documentation.

If the number of values provided is fewer than the number of elements in the array, the unassigned elements are filled with zeros. This case is seen in Figure 8-8(c). We can use this rule to easily initialize an array to all zeros by supplying just the first zero value, as shown in the last example in Figure 8-8(d).

Only fixed-length arrays can be initialized when they are defined. Variable-length arrays must be initialized by inputting or assigning the values.



(a) Basic Initialization

```
int numbers[5] = {3,7,12,24,45};
```

(b) Initialization without Size

```
int numbers[ ] = {3,7,12,24,45};
```

(c) Partial Initialization

```
int numbers[5] = {3,7};
```

(d) Initialization to All Zeros

```
int lotsOfNumbers [1000] = {0};
```

FIGURE 8-8 Initializing Arrays

Inputting Values

Another way to fill the array is to read the values from the keyboard or a file. This method of inputting values can be done using a loop. When the array is to be completely filled, the most appropriate loop is the *for* loop because the number of elements is fixed and known.

```
for (i = 0; i < 10; i++)
    scanf ("%d", &scores [i]);
```

Several concepts need to be studied in this simple statement. First, we start the index, `i`, at 0. Since the array has 9 elements, we must load the values from index locations 0 through 8. The limit test, therefore, is set at `i < 9`, which conveniently is the number of elements in the array. Then, even though we are dealing with array elements, the address operator (`&`) is still necessary in the *scanf* call.

Finally, if all the elements might not be filled, then we use one of the event-controlled loops (*while* or *do...while*). Which loop we use depends on the application.

Assigning Values

We can assign values to individual elements using the assignment operator. Any value that reduces to the proper type can be assigned to an individual array element. A simple assignment statement for `scores` is seen below.

```
scores [4] = 23;
```

On the other hand, we cannot assign one array to another array, even if they match fully in type and size. We have to copy arrays at the individual element level. For example, to copy an array of 25 integers to a second array of 25 integers, we could use a loop, as shown below.

```
for (i = 0; i < 25; i++)
    second[i] = first[i];
```

If the values of an array follow a pattern, we can use a loop to assign values. For example, the following loop assigns a value that is twice the index number to array scores:

```
for (i = 0; i < 10; i++)
    scores [i] = i * 2;
```

For another example, the following code assigns the odd numbers 1 through 17 to the elements of an array named value:

```
for (i = 0; i < 9; i++)
    value [i] = (i * 2) + 1;
```

One array cannot be copied to another using assignment.

Exchanging Values

A common application is to **exchange** the contents of two elements. We see this operation later in the chapter when we talk about sorting arrays. When we exchange variables, we swap the values of elements without knowing what's in them.

For example, imagine we want to swap numbers[3] and numbers[1] in Figure 8-8. A common beginner's mistake would be simply to assign each element to the other, as shown below.

```
numbers [3] = numbers [1];                          // Error
numbers [1] = numbers [3];
```

Although this code looks as if it will do the job, if we trace the code carefully, we find that it does only half the job. numbers[1] is moved to numbers[3], but the second half isn't done. The result is that both elements have the same value. Figure 8-9 traces the steps.

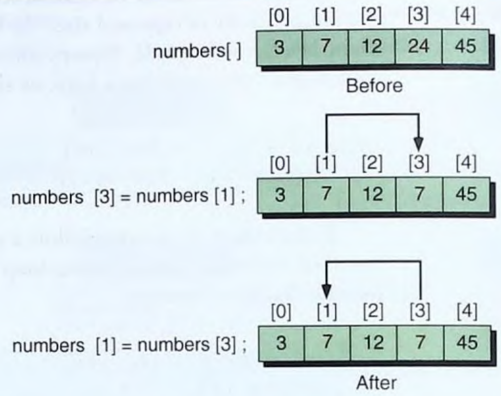


FIGURE 8-9 Exchanging Scores—the Wrong Way

Figure 8-9 shows that the original value of `numbers[3]` is destroyed before we can move it. The solution is to use a temporary variable to store the value in `numbers[3]` before moving the data from `numbers[1]`.

```
temp          = numbers[3];
numbers [3] = numbers [1];
numbers [1] = temp;
```

This technique is shown in Figure 8-10.

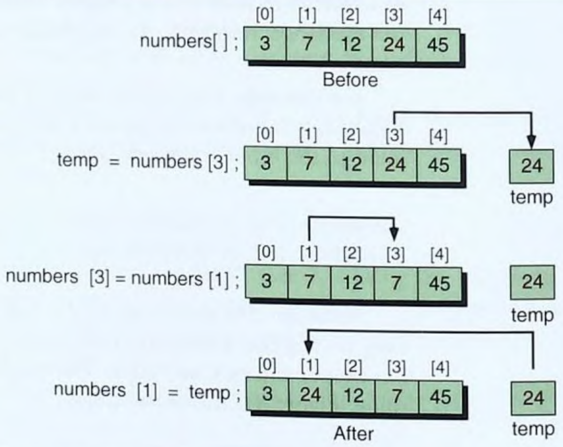


FIGURE 8-10 Exchanging Scores with Temporary Variable

Printing Values

Another common application is printing the contents of an array. This is easily done with a *for* loop, as shown below.

```
for (i = 0; i < 9; i++)
    printf ("%d ", scores [i]);
printf ("\n");
```

In this example, all the data are printed on one line. After the *for* loop completes, a final *printf* statement advances to the next line. But what if we had 100 values to be printed? In that case, we couldn't put them all on one line. Given a relatively small number width, however, we could put 10 on a line. We would then need 10 lines to print all the data. This rather common situation is easily handled by adding a counter to track the number of elements we have printed on one line. This logic is shown in Program 8-1.

PROGRAM 8-1 Print Ten Numbers per Line

```
// a program fragment
#define MAX_SIZE 25

// Local Declarations
    int list [MAX_SIZE];

// Statements
    …
    numPrinted = 0;
    for (int i = 0; i < MAX_SIZE; i++)
        {
         printf("%3d", list[i]);
         if (numPrinted < 9)
              numPrinted++;
         else
             {
              printf("\n");
              numPrinted = 0;
              } // else
        } // for
```

Precedence of Array References

References to elements in arrays are postfix expressions. By looking at the Precedence Table, we see that array references have a priority of 16, which is very high. What is not apparent from the table, however, is that the opening and closing brackets are actually operators. They create a postfix expression from a primary expression. With a little thought, you should recognize that this is exactly as it must be: When we index an array element in an expression, the value must be determined immediately.

Referring to the original array in Figure 8-10, what will be the result of the following code?

```
numbers[3] = numbers[4] + 15;
```

In this case, numbers[4] has a higher precedence (16) than the addition operator (12), so it is evaluated first. The result is then

```
numbers[3] = 45 + 15;
```

After this statement has been executed, numbers[3] has been changed from 24 to 60.

Index Range Checking

The C language does not check the boundary of an array. It is our job as programmers to ensure that all references to indexed elements are valid and within the range of the array. If we use an **invalid index** in an expression, we get unpredictable results. (The results are unpredictable because we have no way of knowing the value of the indexed reference.)

When we use an invalid index in an assignment, we destroy some undetermined portion of the program. Usually, but not always, the program continues to run and either produces unpredictable results or eventually aborts.

An example of a common out-of-range index is seen in the code to fill an array from the keyboard. For this example, we have reproduced the code we used to fill the scores array earlier, only this time we have made a common mistake! Can you spot it?

```
for (i = 1; i <= 9; i++)                                    // Error
   scanf ("%d", &scores [i]);
```

When dealing with array processing, be very careful at the beginning and end of the array. A careful examination of the previous code discloses that we erroneously started at 1 instead of 0. So we fix it as shown below, only to find that it still doesn't work!

```
for (i = 0; i <= 9; i++)
   scanf ("%d", &scores [i]);
```

The moral of this example is to examine our logic. If we made one mistake, we may well have made two. Although we corrected the error for initialization (the beginning of the array), there is still an error at the other end. If you can't see it, check the original code in "Inputting Values."

The result of both versions of this error is that the data stored in memory after the scores array are erroneously destroyed. In the first version of the error, the first element of the array was not initialized.

The problems created by unmanaged indexes are among the most difficult to solve, even with today's powerful programming workbenches. So we want to plan our array logic carefully and fully test it.

EXAMPLE 8-1 Print Array

Let's write a program that uses arrays. Program 8-2 uses a *for* loop to initialize each element in an array to the square of the index value and then prints the array.

PROGRAM 8-2 Squares Array

```c
/* Initialize array with square of index and print it.
      Written by:
      Date:
*/
#include <stdio.h>
#define ARY_SIZE 5

int main (void)
{
// Local Declarations
   int sqrAry[ARY_SIZE];

// Statements
   for (int i = 0; i < ARY_SIZE; i++)
      sqrAry[i] = i * i;

   printf("Element\tSquare\n");
   printf("=======\t======\n");
   for (int i = 0; i < ARY_SIZE; i++)
      printf("%5d\t%4d\n", i, sqrAry[i]);
   return 0;
}  // main
```

```
Results:
Element Square
======= ======
    0       0
    1       1
    2       4
    3       9
    4      16
```

Example: Read and Print Reversed

As another example of an array program, let's read a series of numbers from the keyboard and print them in reverse order; that is, if we read 1 2 3 4, we want to print them 4 3 2 1. With a little thought, it should be obvious that we must read all of the numbers before we can begin printing them. This definitely sounds like a problem for an array.

In Program 8-3, we read up to 50 integers and then print them reversed, 10 to a line.

PROGRAM 8-3 Print Input Reversed

```
/* Read a number series and print it reversed.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int readNum;
   int numbers[50];

// Statements
   printf("You may enter up to 50 integers:\n");
   printf("How many would you like to enter? ");
   scanf ("%d", &readNum);

   if (readNum > 50)
       readNum = 50;

   // Fill the array
   printf("\nEnter your numbers: \n");
   for (int i = 0; i < readNum; i++)
        scanf("%d", &numbers[i]);

   // Print the array
   printf("\nYour numbers reversed are: \n");
   for (int i = readNum - 1, numPrinted = 0;
             i >= 0;
             i--)
       {
        printf("%3d", numbers[i]);
        if (numPrinted < 9)
```

continued

PROGRAM 8-3 Print Input Reversed *(continued)*

```
           numPrinted++;
       else
          {
           printf("\n");
           numPrinted = 0;
          } // else
      } // for
   return 0;
}  // main
```

```
Results:
You may enter up to 50 integers:
How many would you like to enter? 12

Enter your numbers:
1 2 3 4 5 6 7 8 9 10 11 12

Your numbers reversed are:
 12 11 10  9  8  7  6  5  4  3
  2  1
```

Program 8-3 Analysis First, note how we validate the number of integers to be read. If the user requests more than 50, we simply set `readNum` to 50. This ensures that we will not run off the end of the array.

This is an interesting loop because we start at the end and work to the front. While many arrays are processed from the beginning, it is often necessary to process from the end. You will see more in the sorting functions later in the chapter.

Finally, note that we incorporate the logic from Program 8-1 to print ten numbers per line. Had we made Program 8-1 a function, we could have reused the code rather than repeating it.

8.3 Inter-function Communication

To process arrays in a large program, we have to be able to pass them to functions. We can pass arrays in two ways: pass individual elements or pass the whole array. In this section we discuss first how to pass individual elements and then how to pass the whole array.

Passing Individual Elements

As we saw in Chapter 4, we can pass individual elements by either passing their data values or by passing their addresses.

Passing Data Values

We pass data values, that is individual array elements, just like we pass any data value.

As long as the array element type matches the function parameter type, it can be passed. The called function cannot tell whether the value it receives comes from an array, a variable, or an expression. Figure 8-11 demonstrates how an array value and a variable value can be passed to the same function.

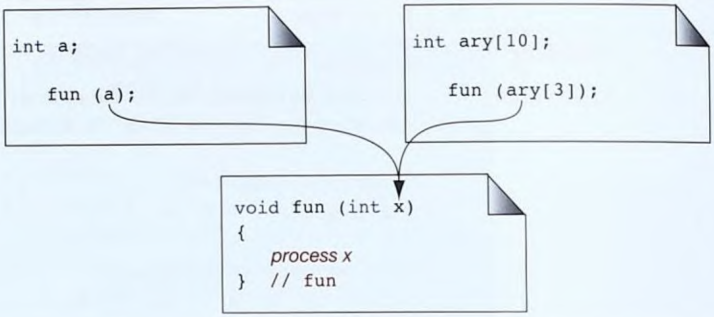


FIGURE 8-11 Passing Array Elements

Note how only one element is passed by using the indexed expression, `ary[3]`. Since the value of this expression is a single integer, it matches the formal parameter type in `fun`. As far as `fun` is concerned, it doesn't know or care that the value it is working with came from an array.

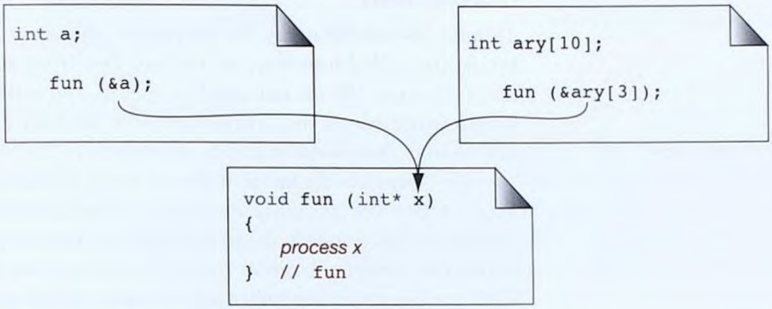
Passing Addresses

In Chapter 4, we saw that we can use two-way communication by passing an address. We can pass the address of an individual element in an array just like we can pass the address of any variable.

To pass an array element's address, we prefix the address operator to the element's indexed reference. Thus, to pass the address of `ary[3]`, we use the code shown in the following example.

```
&ary[3]
```

Passing an address of an array element requires two changes in the called function. First, it must declare that it is receiving an address. Second, it must use the indirection operator (`*`) to refer to the elements value. These concepts are shown in Figure 8-12. Note that when we pass an address, it is two-way communication.



```
int a;

   fun (&a);
```

```
int ary[10];

   fun (&ary[3]);
```

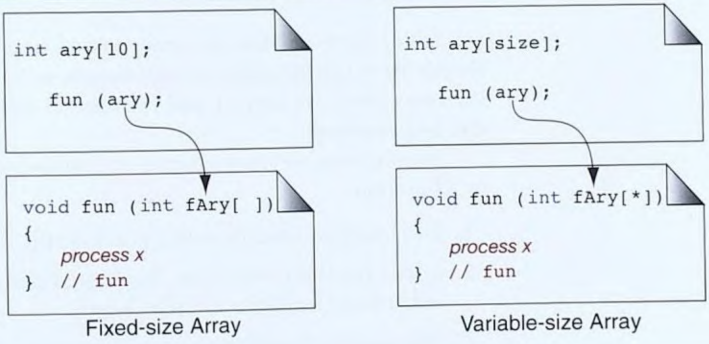
```
void fun (int* x)
{
   process x
}  // fun
```

FIGURE 8-12 Passing the Address of an Array Element

Passing the Whole Array

Here we see the first situation in which C does not pass values to a function. The reason for this change is that a lot of memory and time would be used in passing large arrays every time we wanted to use one in a function. For example, if an array containing 20,000 elements were passed by value to a function, another 20,000 elements would have to be allocated in the called function and each element would have to be copied from one array to the other. So, instead of passing the whole array, C passes the address of the array.

In C, the name of an array is a primary expression whose value is the address of the first element in the array. Since indexed references are simply calculated addresses, all we need to refer to any of the elements in the array is the address of the array. Because the name of the array is in fact its address, passing the array name, as opposed to a single element, allows the called function to refer to the array back in the calling function. The design for passing the whole array is shown in Figure 8-13.



```
int ary[10];

   fun (ary);
```

```
void fun (int fAry[ ])
{
   process x
}  // fun
```

```
int ary[size];

   fun (ary);
```

```
void fun (int fAry[*])
{
   process x
}  // fun
```

FIGURE 8-13 Passing the Whole Array

Fixed-length Arrays

To **pass the whole array**, we simply use the array name as the actual parameter. In the called function, we declare that the corresponding formal parameter is an *array*. We do not need to specify the number of elements in a fixed-length array. Since the array is actually defined elsewhere, all that is important is that the compiler knows it's an array.

The function declaration for an array can declare the array in two ways. First, it can use an array declaration with an empty set of brackets in the parameter list. Second, it can specify that the array parameter is a pointer. We prefer the array declaration over the pointer declaration because it is clearer. Both are shown in the following example, which declares arrays of integers.

```
// Function Declarations
void fun (int ary[], ...);
void fun (int* ary, ...);
```

Variable-length Arrays

When the called function receives a variable-length array, we must declare and define it as variable length. In a function declaration, we declare an array as variable using an asterisk for the array size or by using a variable for the array size. In the function definition, we must use the variable name, and following standard syntax rules, it must be defined before the array. The following example demonstrates these points.

```
// Function Declaration
float calcArrayAvg (int size, float avgArray [*]);
...
// Function Definition
float calcArrayAvg (int size, float avgArray [size])
{
...
} // calcArrayAvg
```

Note, however, that the array passed to a function declaring a variable-length array can be either a fixed-length or variable-length array. As long as the array types are correct and the correct size is specified, the passed array can be processed.

In summary, we must observe the following rules to pass the whole array to a function:

1. The function must be called passing only the name of the array.
2. In the function definition, the formal parameter must be an array type, either fixed length or variable length.
3. The size of a fixed-length array does not need to be specified.
4. The size of a variable-length array in the function prototype must be an asterisk or a variable.

5. The size of a variable-length array in the function definition must be a variable that is in the function's scope, as a previously specified variable parameter.

Passing an Array as a Constant

When a function receives an array and doesn't change it, the array should be received as a constant array. This prevents the function from accidentally changing the array. To declare an array as a constant, we prefix its type with the type qualifier `const`, as shown in the following function definition. Once an array is declared constant, the complier will flag any statement that tries to change it as an error.

```
double average (const int ary[], int size);
```

Function Communication Examples

EXAMPLE 8-2 Average Array Elements

We can use a function to calculate the average of the integers in an array. In this case, we pass the name of the array to the function and it returns the average as a real number. This concept is shown in Program 8-4.

PROGRAM 8-4 Calculate Array Average

```
/* Calculate the average of the number in an array.
      Written by:
      Date:
*/
#include <stdio.h>

// Function Declaration
double average (int ary[ ]);

int main (void)
{
// Local Declarations
   double ave;
   int    base[5] = {3, 7, 2, 4, 5};

// Statement
   ave = average(base);
   printf("Average is: %lf\n", ave);
   return 0;
}  // main

/* =================== average ===================
```

continued

PROGRAM 8-4 Calculate Array Average *(continued)*

```
   Calculate and return average of values in array.
      Pre  Array contains values
      Post Average calculated and returned
*/
double average (int ary[ ])
{
// Local Declarations
   int sum = 0;

// Statement
   for (int i = 0; i < 5; i++)
       sum += ary[i];

   return (sum / 5.0);
}  // average
```

EXAMPLE 8-3 Average Elements in Variable-length Array

In this example, we average the elements in a variable-length array. However, because a variable-length array cannot be initialized, we read the data from the keyboard. Also, note that to keep the program as simple as possible, we create the array in a block. This is necessary because the size of the array must be known before it is declared.

PROGRAM 8-5 Average Elements in Variable-length Array

```
/* Calculate average of numbers in variable-length
   array.
      Written by:
      Date:
*/
#include <stdio.h>

// Function Declaration
double average (int size, int ary[*]);

int main (void)
{
// Local Declarations
   int    size;
   double ave;

// Statements
   printf("How many numbers do you want to average? ");
```

continued

PROGRAM 8-5 Average Elements in Variable-length Array *(continued)*

```
  scanf ("%d", &size);

  // Create and fill variable-length array
  {
   // Local Declaration
   int ary[size];

   // Fill array
   for (int i = 0; i < size; i++)
      {
       printf("Enter number %2d: ", i + 1);
       scanf ("%d", &ary[i]);
       } // for
    ave = average(size, ary);
  } // Fill array block

  printf("Average is: %lf", ave);
  return 0;
}  // main

/* ==================== average ====================
   Calculate and return average of values in array.
      Pre  Array contains values
      Post Average calculated and returned
*/
double average (int size, int ary[size])
{
// Local Declarations
   int sum = 0;
   double ave;

// Statement
   for (int i = 0; i < size; i++)
       sum += ary[i];

   ave = (double)sum / size;
   return ave;
}  // average
```

```
Results
How many numbers do you want to average? 5
Enter number  1: 3
Enter number  2: 6
Enter number  3: 9
```

continued

PROGRAM 8-5 Average Elements in Variable-length Array *(continued)*

```
      Enter number  4: 2
      Enter number  5: 8
      Average is: 5.600000
```

Program 8-5 Analysis There are two points to study in this program. First, in *main*, we use a variable-length array. This requires that the array size be known before the array is declared. We accomplish this by using a block within *main*. The array size is declared in *main*'s local declarations; the array itself is declared in the block local declarations.

Second, note how the variable array is declared in the function definition. We can do it by just declaring it as a variable array with an asterisk in the array brackets, or we can put the variable `size` in the brackets. Using `size` is better documentation because it declares our intent. When we use `size`, however, it must be declared before the array, in this case as the first parameter.

EXAMPLE 8-4 Changing Values in an Array

The previous examples were one-way communication. In Program 8-6 we demonstrate two-way communication by changing the values in the array. As you study it, note that no special code is required when we want to change the values in an array. Because the array's address is passed, we can simply use index notation to change it.

PROGRAM 8-6 Change Values in an Array

```
/* Multiply each element in an array by 2.
      Written by:
      Date:
*/
#include <stdio.h>
// Function Declaration
void multiply2 (int x[ ]);

int main (void)
{
// Local Declarations
   int base[5] = {3, 7, 2, 4, 5};

// Statements
   multiply2 (base);

   printf("Array now contains: ");
   for (int i = 0; i < 5; i++)
      printf("%3d", base[i]);
   printf("\n");
   return 0;
```

continued

PROGRAM 8-6 Change Values in an Array

```
} // main

/* ==================== multiply2 ====================
   Multiply each element in array by 2.
      Pre  array contains data
      Post each element doubled
*/
void multiply2 (int ary[ ])
{
// Statements
   for (int i = 0; i < 5; i++)
       ary[i] *= 2;
   return;
}  // multiply2
```

8.4 Array Applications

In this section we study two array applications: frequency arrays with their graphical representations and random number permutations.

Frequency Arrays

Two common statistical applications that use arrays are frequency distributions and histograms. A **frequency array** shows the number of elements with an identical value found in a series of numbers. For example, suppose we have taken a sample of 100 values between 0 and 19. We want to know how many of the values are 0, how many are 1, how many are 2, and so forth up through 19.

We can read these numbers into an array called `numbers`. Then we create an array of 20 elements that will show the frequency of each number in the series. This design is shown in Figure 8-14.

With the data structure shown in Figure 8-14 in mind, how do we write the application? Since we know that there are exactly 100 elements, we can use a *for* loop to examine each value in the array. But how can we relate the value in `numbers` to a location in the frequency?

One way to do it is to assign the value from the data array to an index and then use the index to access the frequency array. This technique is shown below.

```
f = numbers[i];
frequency [f]++;
```

Since an index is an expression, however, we can simply use the value from our data array to index us into the frequency array as shown in the next example. The value of numbers[i] is determined first, and then that value is used to index into frequency.

```
frequency [numbers [i]]++;
```

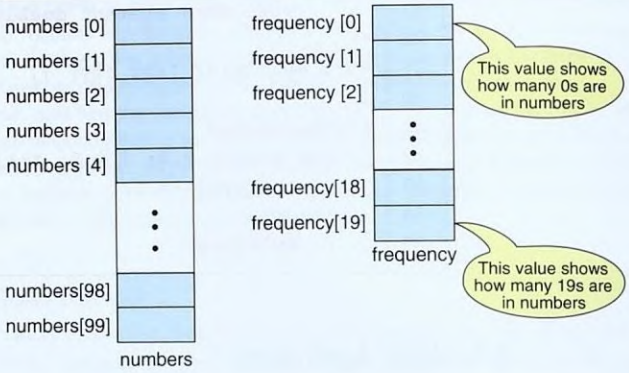
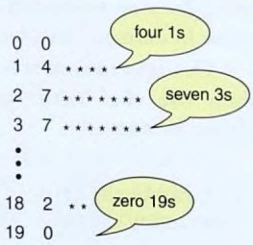


FIGURE 8-14 Frequency Array

The complete function is shown in Program 8-7 as makeFrequency. The function first initializes the frequency array and then scans the data array to count the number of occurrences of each value.

Histograms

A **histogram** is a pictorial representation of a frequency array. Instead of printing the values of the elements to show the frequency of each number, we print a histogram in the form of a bar chart. For example, Figure 8-15 is a histogram for a set of numbers in the range 0…19. In this example, asterisks (*) are used to build the bar. Each asterisk represents one occurrence of the data value.



```
 0  0
 1  4 ****
 2  7 *******
 3  7 *******
 :
18  2 **
19  0
```

FIGURE 8-15 Frequency Histogram

Let's write a program that builds a frequency array for data values in the range 0…19 and then prints their histogram. The data are read from a file. To provide flexibility, the `getData` function may only partially fill the array. The function that loads it also guards against too much data. The design for the program is shown in Figure 8-16.

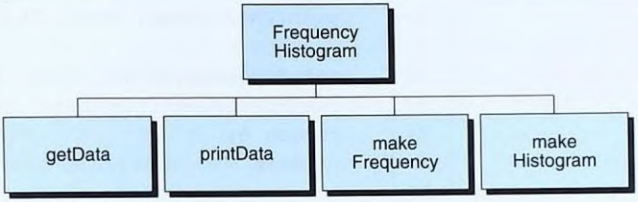


FIGURE 8-16 Histogram Program Design

The code is shown in Program 8-7.

PROGRAM 8-7 Frequency and Histogram

```
/* Read data from a file into an array.
   Build frequency array & print data in histogram.
      Written by:
      Data:
*/
#include <stdio.h>
#include <stdlib.h>

#define MAX_ELMNTS 100
#define ANLYS_RNG 20

// Function Declarations
   int getData (int numbers[], int size, int range);

   void printData     (int numbers[],
                       int size,        int lineSize);
   void makeFrequency (int numbers[],   int size,
                       int frequency[], int range);
   void makeHistogram (int frequency[], int range);

int main (void)
{
// Local Declarations
   int size;
```

continued

PROGRAM 8-7 Frequency and Histogram *(continued)*

```
   int nums      [MAX_ELMNTS];
   int frequency [ANLYS_RNG];

// Statements
   size = getData (nums, MAX_ELMNTS, ANLYS_RNG);
   printData (nums, size, 10);

   makeFrequency(nums, size, frequency, ANLYS_RNG);
   makeHistogram(frequency, ANLYS_RNG);
   return 0;
}  // main

/* ==================== getData ====================
   Read data from file into array. The array
   does not have to be completely filled.
      Pre   data is an empty array
            size is maximum elements in array
            range is highest value allowed
      Post  array is filled. Return number of elements
*/
int getData (int data [], int size, int range)
{
// Local Declarations
   int   dataIn;
   int   loader = 0;
   FILE* fpData;



// Statements
   if (!(fpData = fopen ("P08-07.dat", "r")))
      printf("Error opening file\a\a\n") , exit (100);

   while (loader < size
        && fscanf(fpData, "%d", &dataIn) != EOF)
      if (dataIn >= 0 && dataIn < range)
          data[loader++] = dataIn;
      else
         printf("\nData point %d invalid. Ignored. \n",
                 dataIn);

// Test to see what stopped while
   if (loader == size)
      printf("\nToo much data. Process what read.\n");
```

continued

PROGRAM 8-7 Frequency and Histogram *(continued)*

```
   return loader;
}  // getData

/* ==================== printData ====================
   Prints the data as a two-dimensional array.
      Pre   data: a filled array
            size: number of elements in array
            lineSize: number of elements printed/line
      Post  the data have been printed
*/
void printData (int data[], int size, int lineSize)
{
// Local Declarations
   int numPrinted = 0;

// Statements
   printf("\n\n");
   for (int i = 0; i < size; i++)
      {
       numPrinted++;
       printf("%2d ", data[i]);
       if (numPrinted >= lineSize)
          {
           printf("\n");
           numPrinted = 0;
          } // if
      } // for
   printf("\n\n");
   return;
}  // printData

/* ================ makeFrequency ==================
   analyze the data in nums and build their frequency
   distribution
      Pre   nums: array of validated data to be analyzed
            last: number of elements in array
            frequency: array for accumulation.
            range: maximum index/value for frequency
      Post  Frequency array has been built.
*/
void makeFrequency (int nums[],      int last,
                    int frequency[], int range)
{
// Statements
```

continued

PROGRAM 8-7 Frequency and Histogram *(continued)*

```
    // First initialize the frequency array
    for (int f = 0; f < range; f++)
        frequency [f] = 0;

    // Scan numbers and build frequency array
    for (int i = 0; i < last; i++)
        frequency [nums [i]]++;
    return;
}  // makeFrequency

/* ================ makeHistogram =================
    Print a histogram representing analyzed data.
       Pre   freq contains times each value occurred
             size represents elements in frequency array
       Post  histogram has been printed
*/
void makeHistogram (int freq[], int range)
{
// Statements
    for (int i = 0; i < range; i++)
       {
        printf ("%2d %2d ", i, freq[i]);
        for (int j = 1; j <= freq[i]; j++)
            printf ("*");
        printf ("\n");
       } // for i...
    return;
}  // makeHistogram
// ================ End of Program ================
```

```
Results:
Data point 20 invalid. Ignored.

Data point 25 invalid. Ignored.

  1  2  3  4  5  6  7  8  7 10
  2 12 13 13 15 16 17 18 17  7
  3  4  6  8 10  2  4  6  8 10
  4  3  5  7  1  3  7  7 11 13
  5 10 11 12 13 16 18 11 12  7
  6  1  2  2  3  3  3  4  4  4
  7  7  8  7  6  5  4  1  2  2
  8 11 11 13 13 13 17 17  7  7
```

continued

PROGRAM 8-7 Frequency and Histogram *(continued)*

```
        13 17 17 15 15

         0  0
         1  4 ****
         2  7 *******
         3  7 *******
         4  8 ********
         5  4 ****
         6  5 *****
         7 12 ************
         8  5 *****
         9  0
        10  4 ****
        11  5 *****
        12  3 ***
        13  8 ********
        14  0
        15  3 ***
        16  2 **
        17  6 ******
        18  2 **
        19  0
```

Program 8-7 Analysis Remember our discussion of what happens when an index gets out of range? What if one of the numbers in our data is greater than 19? We would destroy some other part of our program! To protect against this possibility, we test each data value to make sure that it is within the indexing range of `frequency`. If it is not, we display an error message and read the next number.

Similarly, we guard against too much data. If the file contains more than 100 valid numbers, then we stop reading. After the read loop, we test to make sure we are processing all of the data. If there were too many numbers, we print an error message.

Finally, note that we print the data to make it easy to verify the results.

Random Number Permutations

A random number permutation is a set of random numbers in which no numbers are repeated. For example, given a random number permutation of 10 numbers, the values from 0 to 9 would all be included with no duplicates.

We saw how to generate a set of random numbers in Chapter 4. To generate a permutation, we need to eliminate the duplicates. Borrowing from the histogram concept in the previous section, we can solve the problem most efficiently by using two arrays. The first array contains the random numbers. The second array contains a logical value that indicates whether or not the number represented by its index has been placed in the random number array. This design is shown in Figure 8-17.

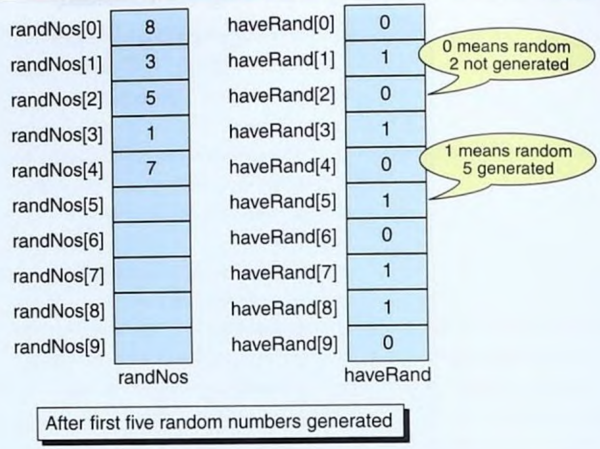


FIGURE 8-17 Design for random number permutations

As you study Figure 8-17, note that only the first five random numbers have been placed in the permutation. For each random number in the random number array, its corresponding location in the have-random array is set to 1. Those locations representing numbers that have not yet been generated are still set to 0. The implementation of the design is shown in Program 8-8[1].

PROGRAM 8-8 Generate a Permutation

```
/* Generate a random number permutation.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>

#define  ARY_SIZE  20

// Function Declarations
void bldPerm   (int randNos[]);
void printData (int data[], int size, int lineSize);

int main (void)
```

continued

[1] The numbers generated by this program are a repeating series. To make them different with each run, we would use the `srand` function and set it to the time of day as explained in Chapter 4.

PROGRAM 8-8 Generate a Permutation *(continued)*

```
{
// Local Declarations
   int randNos [ARY_SIZE];

// Statements
   printf("Begin Random Permutation Generation\n");

   bldPerm   (randNos);
   printData (randNos, ARY_SIZE, 10);

   return 0;
}  // main

/* ==================== bldPerm ====================
   Generate a random number permutation in array.
      Pre   randNos is array to receive permutations
      Post  randNos filled
*/
void bldPerm (int randNos[])
{
// Local Declarations
   int oneRandNo;
   int haveRand[ARY_SIZE] = {0};

// Statements
   for (int i = 0; i < ARY_SIZE; i++)
       {
        do
          {
           oneRandNo = rand() % ARY_SIZE;
          } while (haveRand[oneRandNo] == 1);
        haveRand[oneRandNo] = 1;
        randNos[i] = oneRandNo;
       } // for
   return;
}  // bldPerm

/* ==================== printData ====================
   Prints the data as a two-dimensional array.
      Pre   data: a filled array
            last: index to last element to be printed
            lineSize: number of elements on a line
      Post  data printed
*/
```

continued

PROGRAM 8-8 Generate a Permutation *(continued)*

```
void printData (int data[], int size, int lineSize)
{
// Local Declarations
   int numPrinted = 0;

// Statements
   printf("\n");
   for (int i = 0; i < size; i++)
      {
       numPrinted++;
       printf("%2d ", data[i]);
       if (numPrinted >= lineSize)
          {
           printf("\n");
           numPrinted = 0;
          } // if
      } // for
   printf("\n");
   return;
}  // printData
```

```
Results:
Begin Random Permutation Generation

18 13 15 11  7 10 19 12  6  9
 4  0  5  3 17 14  2 16  1  8
```

8.5 Sorting

One of the most common applications in computer science is sorting—the process through which data are arranged according to their values. We are surrounded by data. If the data are not ordered, we would spend hours trying to find a single piece of information. Imagine the difficulty of finding someone's telephone number in a telephone book that is not ordered in name sequence!

In this chapter we introduce three sorting algorithms: the selection sort, bubble sort, and insertion sort. In each section we will first introduce the basic concept, then use the idea in an example, and finally develop the code for the algorithm.

One programming concept common to the sorting algorithms we discuss in this section is the swapping of data between two elements in a list. You might want to review our discussion in "Exchanging Values" in Section 8.2.

Selection Sort

In the **selection sort**, the list is divided into two sublists, sorted and unsorted, which are divided by an imaginary wall. We find the smallest element from the unsorted sublist and swap it with the element at the beginning of the unsorted sublist. After each selection and swapping, the wall between the two sublists moves one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones. Each time we move one element from the unsorted sublist to the sorted sublist, we say that we have completed a **sort pass**. If we have a list of *n* elements, we need *n* – 1 passes to completely rearrange the data. The selection sort is graphically presented in Figure 8-18.

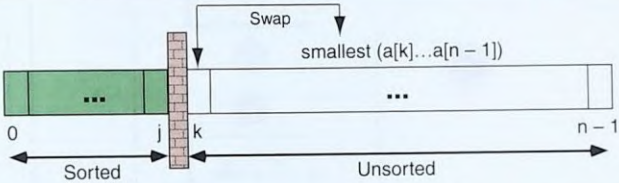


FIGURE 8-18 Selection Sort Concept

Figure 8-19 traces a set of six integers as we sort them. It shows how the wall between the sorted and unsorted sublists moves in each pass. As you study the figure, you will see that the array is sorted after five passes, which is one fewer than the number of elements in the array. This means that our sort loop has one less iteration than the number of elements in the array.

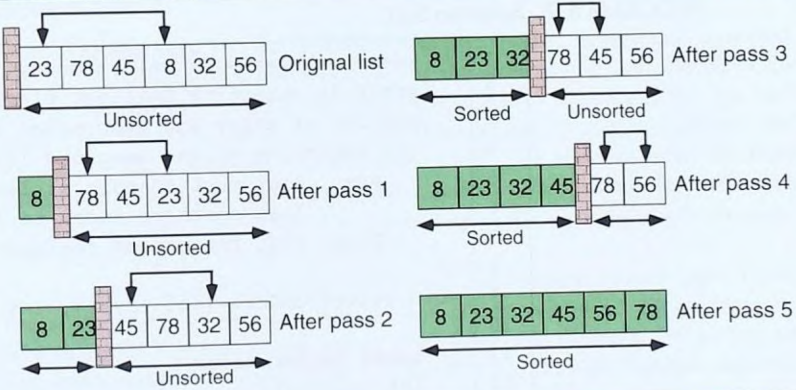


FIGURE 8-19 Selection Sort Example

The program design is a rather straightforward algorithm. Starting with the first item in the list, it uses an inner loop to examine the unsorted items in the list for the smallest element. In each pass, the smallest element is "selected" and exchanged with the first unsorted element. The outer loop is repeated until the list is completely sorted. The design is shown in Figure 8-20.

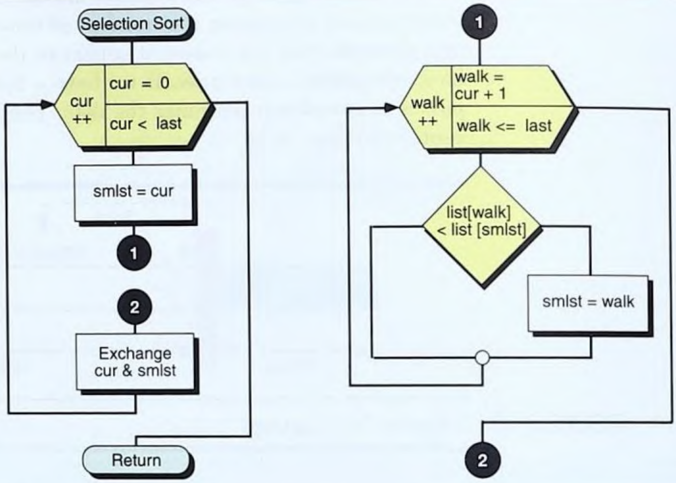


FIGURE 8-20 Design for Selection Sort

The code is shown in Program 8-9. We have highlighted the inner loop to make it easy to follow the logic.

PROGRAM 8-9 Selection Sort

```
/* ================== selectionSort ===================
   Sorts by selecting smallest element in unsorted
   portion of array and exchanging it with element at
   the beginning of the unsorted list.
      Pre   list must contain at least one item
            last contains index to last element in list
      Post  list rearranged smallest to largest
*/
void selectionSort (int list[], int last)
{
// Local Declarations
   int smallest;
   int tempData;

```

continued

Selection Sort *(continued)*

```
 // Statements
    // Outer Loop
    for (int current = 0; current < last; current++)
          {
           smallest = current;
           // Inner Loop: One sort pass each loop
           for (int walk = current + 1;
                    walk <= last;
                    walk++)
                 if (list[walk] < list[smallest])
                      smallest = walk;
          // Smallest selected: exchange with current
          tempData       = list[current];
          list[current]  = list[smallest];
          list[smallest] = tempData;
          } // for current
     return;
 }   // selectionSort
```

Program 8-9 Analysis In this algorithm we see two elements that are common to all three sorts discussed in this section. First, each algorithm makes use of an inner loop either to determine the proper location of the next element or to identify and exchange two elements. Each iteration of the inner loop is one sort pass.

Second, each time we need to move data, we must use a temporary storage area. This technique is found in every sort algorithm except those that use two sorting areas. In the selection sort, the temporary area is used to exchange the two elements.

Bubble Sort

In the **bubble sort**, the list is divided into two sublists, sorted and unsorted. The smallest element is *bubbled* from the unsorted sublist and moved to the sorted sublist. After moving the smallest element to the sorted list, the wall moves one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones. Each time an element moves from the unsorted sublist to the sorted sublist, one sort pass is completed. Given a list of n elements, the bubble sort requires up to $n - 1$ passes to sort the data. The bubble concept is seen in Figure 8-21.

Figure 8-22 shows how the wall moves one element in each pass. Looking at the first pass, we start with 56 and compare it to 32. Since 56 is not less than 32, it is not moved and we step down one element. No exchanges take place until we compare 45 to 8. Since 8 is less than 45, the two elements are exchanged and we step down 1 element. Because 8 was moved down, it is now compared to 78 and these two elements are exchanged. Finally, 8 is compared to 23 and exchanged. This series of exchanges places 8 in the first location and the wall is moved up one position.

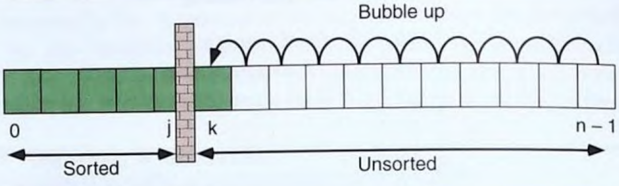


FIGURE 8-21 Bubble Sort Concept

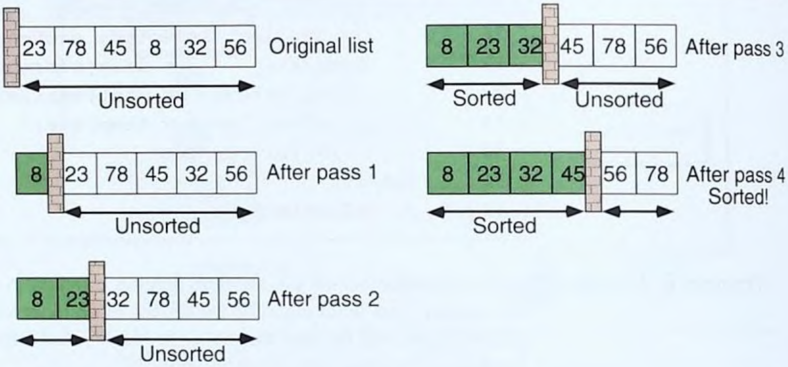

23 78 45 8 32 56 Original list
Unsorted
8 23 78 45 32 56 After pass 1
Unsorted
8 23 32 78 45 56 After pass 2
Unsorted
8 23 32 45 78 56 After pass 3
Sorted
Unsorted
8 23 32 45 56 78 After pass 4
Sorted!
Sorted

FIGURE 8-22 Bubble Sort Example

The bubble sort was originally written to "bubble up" the highest element in the list. From an efficiency point of view, it makes no difference whether the high element is bubbled or the low element is bubbled. From a consistency point of view, however, it makes comparisons between the sorts easier if all three of them work in the same manner. For that reason, we have chosen to bubble the lowest key in each pass.

Like the selection sort, the bubble sort is quite simple. In each pass through the data, controlled by the outer *for* loop, the lowest element is *bubbled* to the beginning of the unsorted segment of the array. The bubbling process is done in the inner loop.

Each time it is executed, the inner *for* loop makes one pass through the data. Whenever it finds two elements out of sequence, it exchanges them. It then continues with the next element. This process allows the smallest element to be bubbled to the beginning of the array, while at the same time adjacent elements along the way are rearranged. The design is shown in Figure 8-23.

The bubble sort is shown in Program 8-10.

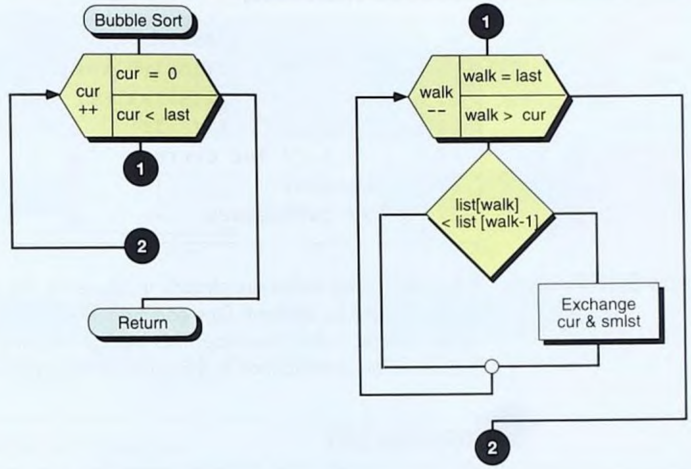


FIGURE 8-23 Bubble Sort Design

PROGRAM 8-10 Bubble Sort

```
/* ==================== bubbleSort ====================
   Sort list using bubble sort. Adjacent elements are
   compared and exchanged until list is ordered.
      Pre  the list must contain at least one item
           last contains index to last element in list
      Post list rearranged in sequence low to high
*/
void bubbleSort (int list [], int last)
{
// Local Declarations
   int temp;

// Statements
   // Outer loop
   for(int current = 0; current < last; current++)
      {
       // Inner loop: Bubble up one element each pass
       for (int walker = last;
                walker > current;
                walker--)
            if (list[walker]  < list[walker - 1])
               {
```

continued

PROGRAM 8-10 Bubble Sort *(continued)*

```
               temp             = list[walker];
               list[walker]     = list[walker - 1];
               list[walker - 1] = temp;
              } // if
      } // for current
   return;
}  // bubbleSort
```

Program 8-10 Analysis If the data being sorted are already in sequence, the inner loop still goes through the array element by element. One common modification is to stop the sort if there are no exchanges in the inner loop. This change requires the addition of a "sorted" flag. We leave this modification to the practice sets at the end of the chapter.

Insertion Sort

The **insertion sort** algorithm is one of the most common sorting techniques used by card players. As they pick up each card, they insert it into the proper sequence in their hand.[2]

In the insertion sort, the list is divided into two parts: sorted and unsorted. In each pass, the first element of the unsorted sublist is picked up and transferred into the sorted sublist by inserting it at the appropriate place. If we have a list of n elements, it will take at most $n - 1$ passes to sort the data (Figure 8-24).

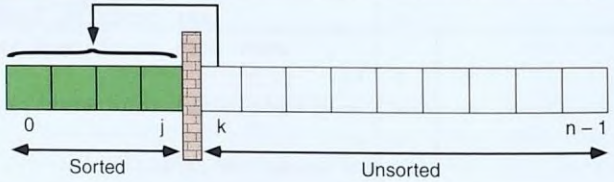


FIGURE 8-24 Insertion Sort Concept

Figure 8-25 traces the insertion sort through our list of six numbers. Each pass moves the wall as an element is removed from the unsorted sublist and inserted into the sorted sublists.

[2] As an aside, card sorting is an example of a sort that uses two pieces of data to sort: suit and rank.

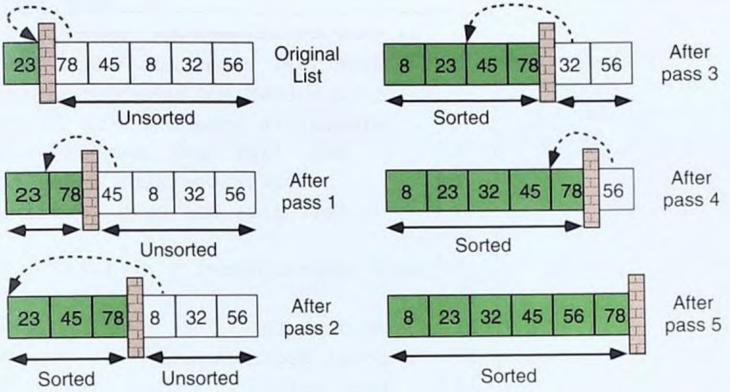


FIGURE 8-25 Insertion Sort Example

The design of the insertion sort follows the same pattern we saw in both the selection sort and the bubble sort; each iteration of the outer loop is a sort pass. The inner loop inserts the first element from the unsorted list into its proper position relative to the rest of the data in the sorted list. The design is shown in Figure 8-26.

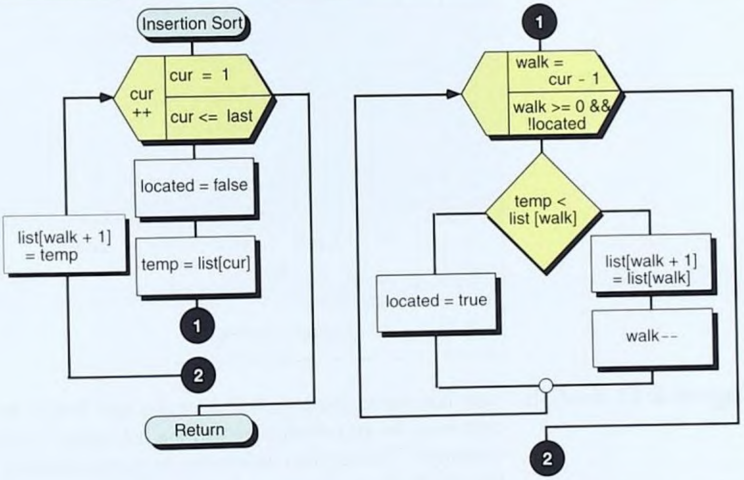


FIGURE 8-26 Insertion Sort Design

Program 8-11 shows the insertion sort.

PROGRAM 8-11 Insertion Sort

```
/* ================= insertionSort =================
   Sort list using Insertion Sort. The list is divided
   into sorted and unsorted lists. With each pass, first
   element in unsorted list is inserted into sorted list.
      Pre  list must contain at least one element
           last contains index to last element in list
      Post list has been rearranged
*/
void insertionSort (int list[], int last)
{
// Statements
// Local Declarations
   int  walk;
   int  temp;
   bool located;

// Statements
   // Outer loop
   for (int current = 1; current <= last; current++)
      {
       // Inner loop: Select and move one element
       located = false;
       temp = list[current];
       for (walk = current - 1; walk >= 0 && !located;)
          if (temp < list[walk])
             {
              list[walk + 1] = list[walk];
              walk--;
             } // if
          else
              located = true;
       list [walk + 1] = temp;
      } // for
    return;
 }  // insertionSort
```

Program 8-11 Analysis Note how the exchange is worked in this sort. Before the inner loop starts, we put the data from the current element into the hold area (`temp`). This is the first step in the exchange. The loop then determines the correct position to place the element by starting with the largest element in the sorted list and working toward the beginning. As it searches, it spreads the sorted portion of the list by shifting each element one position higher in the list. When it locates the correct position, therefore, the data have already been moved right one position and the current location is "empty"; the sort therefore simply places the saved element in its proper location, completing the exchange.

Testing Sorts

Now that we have written three sorts, how do we test them? The answer is that we write a simple test driver. The same test driver can be used for all three sorts. The only changes necessary are in the include statement, the prototype statement, and the function call (see highlighted statements). Program 8-12 is set up to test the insertion sort.

PROGRAM 8-12 Testing Sorts

```c
/* Test driver for insertion sort.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdbool.h>
#include "P08-11.c"

#define MAX_ARY_SIZE 15

// Function Declarations
void insertionSort  (int list[], int last);

int main (void)
{
// Local Declarations
   int ary[MAX_ARY_SIZE] = { 89, 72,  3, 15, 21,
                             57, 61, 44, 19, 98,
                              5, 77, 39, 59, 61 };

// Statements
   printf("Unsorted: ");
   for (int i = 0; i < MAX_ARY_SIZE; i++)
        printf("%3d", ary[i]);

   insertionSort (ary, MAX_ARY_SIZE - 1);

   printf("\nSorted  : ");
   for (int i = 0; i < MAX_ARY_SIZE; i++)
        printf("%3d", ary[i]);
   printf("\n");
   return 0;
}  // main
```

```
Results:
Unsorted:  89 72  3 15 21 57 61 44 19 98  5 77 39 59 61
Sorted  :   3  5 15 19 21 39 44 57 59 61 61 72 77 89 98
```

Sorts Compared

It is interesting to compare the three sorting algorithms. All three use two nested loops. The outer loop is the same in all three; it moves the wall one element toward the beginning of the array. The processing in the inner loop, however, changes with each sort.

In the selection sort, the inner loop starts at the unsorted portion of the array and finds the smallest element in the unsorted area. Once located, it swaps the smallest element with the first unsorted element. It then moves the wall and continues the sort. Note that all exchanges take place in the unsorted portion of the array; the sorted elements are never touched.

While the selection sort starts at the beginning of the unsorted portion of the array, the bubble sort starts at the end. Another difference between the two is that the bubble sort may make multiple exchanges in each pass. While bubbling its way to the beginning of the array, it exchanges all elements that are out of sequence. Like the selection sort, once the wall is moved, the sorted data is never touched.

The insertion sort takes a completely different approach. In each pass, the first unsorted element is inserted into the sorted portion of the array. While it might appear that there are no exchanges in the insertion sort, there are. As explained previously, the first unsorted element is moved to a hold area. This is the first step in an exchange. Then, as it loops through the unsorted portion of the array, elements are shifted higher. Each shift is the equivalent of the second step in an exchange. Finally, when the correct location is found, the third step of the exchange moves the temporarily stored element to its correct position.

Table 8-1 summarizes the exchange and shift characteristics of these three sorts.

Sort	Exchanges	Shifts
Selection	Once for each pass	
Bubble	Several in each pass	
Insertion	Only partially in loop	Zero or more in each pass

TABLE 8-1 Sort Exchanges and Passes

Sort Conclusions

In this section, we have covered three classic sorts. With the exception of the insertion sort, you generally will not find them implemented in production systems. The insertion sort is used as a subfunction in both Quicksort and

Singleton's variation, Quickersort, which are considered the best general-purpose sorts.[3]

Historically, however, these three sorts are the foundation of improved and faster sorting methods that you will study in a data structures course. The selection sort is the foundation of a sorting method called the heap sort; the bubble sort is the foundation for Quicksort and Quickersort; and the insertion sort is the foundation of a sorting method called Shell Sort.

8.6 Searching

Another common operation in computer science is **searching**, which is the process used to find the location of a target among a list of objects. In the case of an array, searching means that given a value, we want to find the location (index) of the first element in the array that contains that value. The search concept is shown in Figure 8-27.

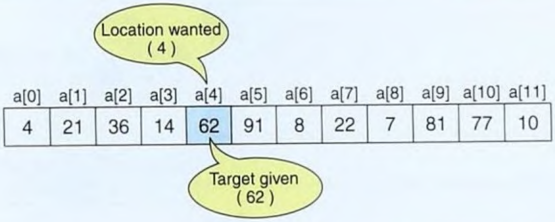


FIGURE 8-27 Search Concept

The algorithm used to search a list depends to a large extent on the structure of the list. Since our structure is currently limited to arrays, we will study searches that work with arrays. As we study other structures, we will study different search techniques.

There are two basic searches for arrays: the sequential search and the binary search. The sequential search can be used to locate an item in any array. The binary search, on the other hand, requires the list to be sorted.

Sequential Search

The **sequential search** is used whenever the list is not ordered. Generally, we use the technique only for small lists or lists that are not searched often. In

[3] For a discussion of Quicksort, see Richard F. Gilberg and Behrouz A. Forouzan, *Data Structures: A Pseudocode Approach With C*. Pacific Grove, CA: Brooks/Cole, 1998.

other cases we would first sort the list and then search it using the binary search discussed later.

In the sequential search, we start searching for the target from the beginning of the list, and we continue until we find the target or until we are sure that it is not in the list. This gives us two possibilities; either we find it or we reach the end of the list. Figure 8-28 traces the steps to find the value 62. We first check the data at index 0, then 1, 2, and 3 before finding the 62 in the fifth element (index 4).

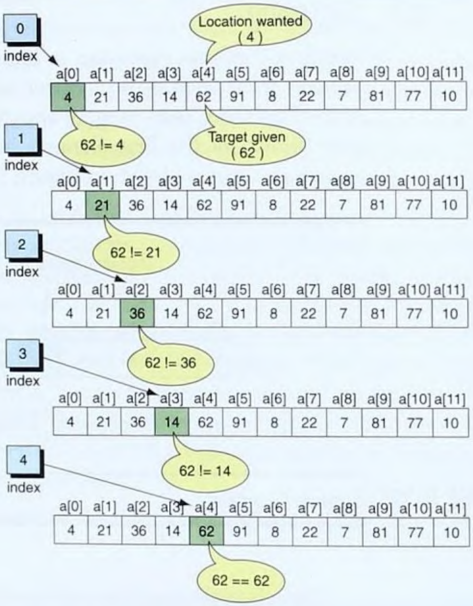


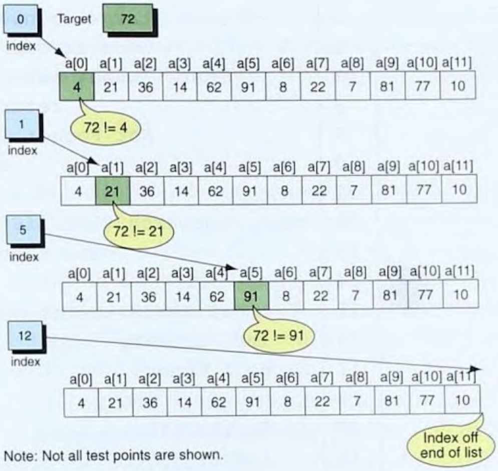
FIGURE 8-28 Locating Data in Unordered List

But what if the target is not in the list? Then we have to examine each element until we reach the end of the list. Figure 8-29 traces the search for a target of 72. When we detect the end of the list, we know that the target does not exist.

Let's write the sequential search function. A search function needs to tell the calling function two things: Did it find the data it was looking for? If it did, what is the index at which the data were found?

But a function can return only one value. For search functions, we use the return value to designate whether we found the target or not. To "return" the index location where the data were found, we will use call-by-address.

The search function requires four parameters: the list we are searching, the index to the last element in the list, the target, and the address where the found element's index location will be stored. Although we could write it without passing the index to the last element, that would mean the search would have to know how many elements are in the list. To make the function as flexible as possible, therefore, we pass the index of the last data value in the array. This is also a good structured design technique. With this information, we are now ready to create the design. It is shown in Figure 8-30.



Note: Not all test points are shown.

FIGURE 8-29 Unsuccessful Search in Unordered List

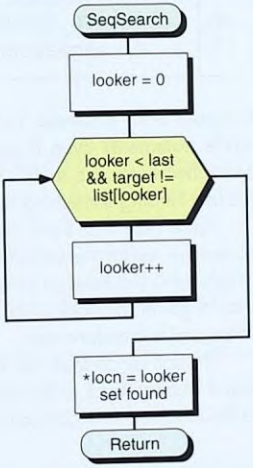


FIGURE 8-30 Sequential Search Design

The code is shown in Program 8-13.

PROGRAM 8-13 Sequential Search

```
/* ==================== seqSearch ====================
   Locate target in an unordered list of size elements.
      Pre   list must contain at least one item
            last is index to last element in list
            target contains the data to be located
            locn is address for located target index
      Post  Found: matching index stored in locn
                   return true (found)
            Not Found: last stored in locn
                       return false (not found)
*/
bool seqSearch (int list[], int  last,
                int target, int* locn)
{
// Local Declarations
   int  looker;
   bool found;

// Statements
   looker = 0;
   while (looker < last && target != list[looker])
      looker++;

   *locn = looker;
   found = (target == list[looker]);
   return found;
} // seqSearch
```

Program 8-13 Analysis

Program 8-13 is simple, but it does merit some discussion. First, why did we use a *while* statement? Even though we know the limits of the array, it is still an event-controlled loop. We search until we find what we are looking for or reach the end of the list. Finding something is an event, so we use an event loop.

Next, note that there are two tests in the limit expression of the loop. We have coded the test for the end of the array first. In this case, it doesn't make any difference which test is first from an execution point of view, but in other search loops it might. You should get in the habit of coding the limit test first because it doesn't use an indexed value and is therefore safer.

The call-by-address use for `locn` also merits discussion. Since we need to pass the found location back to the variable in the calling program, we need to pass its address to the function. A typical call to the search would look like the statement shown below.

```
found = seqSearch (stuAry, lastStu, stuID, &locn);
```

Notice how succinct this function is. In fact, there are more lines of documentation than there are lines of code. The entire search is contained in one *while* statement. With this short code, you might be tempted to ask, "Why write the function at all? Why not just put the one line of code wherever it is needed?" The answer lies in the structured programming concepts that each function should do only one thing and in the concept of reusability. By isolating the search process in its own function, we separate it from the process that needs the search. This is better structured programming. This also makes the code **reusable** in other parts of the program and portable to other programs that need searching.

One final point: This function assumes that the list is not ordered. If the list were ordered, we could improve the search slightly when the data we were looking for were not in the list. We will leave this improvement for a problem at the end of the chapter.

Binary Search

The sequential search algorithm is very slow. If we have an array of 1 million elements, we must do 1 million comparisons in the worst case. If the array is not sorted, this is the only solution. But if the array is sorted, we can use a more efficient algorithm called the **binary search**. Generally speaking, we should use a binary search whenever the list starts to become large. The definition of large is vague. We suggest that you consider binary searches whenever the list contains more than 50 elements.

The binary search starts by testing the data in the element at the middle of the array. This determines if the target is in the first half or the second half of the list. If it is in the first half, we do not need to check the second half. If it is in the second half, we don't need to test the first half. In other words, either way we eliminate half the list from further consideration. We repeat this process until we find the target or satisfy ourselves that it is not in the list.

To find the middle of the list, we need three variables, one to identify the beginning of the list, one to identify the middle of the list, and one to identify the end of the list. We will analyze two cases: the target is in the list, and the target is not in the list.

Target Found

Figure 8-31 shows how we find 22 in a sorted array. We descriptively call our three indexes first, mid, and last. Given first as 0 and last as 11, we can calculate mid as follows:[4]

```
mid = (first + last) / 2;
```

Since the index mid is an integer, the result will be the integral value of the quotient; that is, it truncates rather than rounds the calculation. Given the data in Figure 8-31, mid becomes 5 as a result of the first calculation.

```
mid = (0 + 11) / 2 = 11 / 2 = 5
```

[4] This formula does not work if the number of elements in the array is greater than half MAX_INT. In that case, the correct formula is: mid = first + (last – first) / 2.

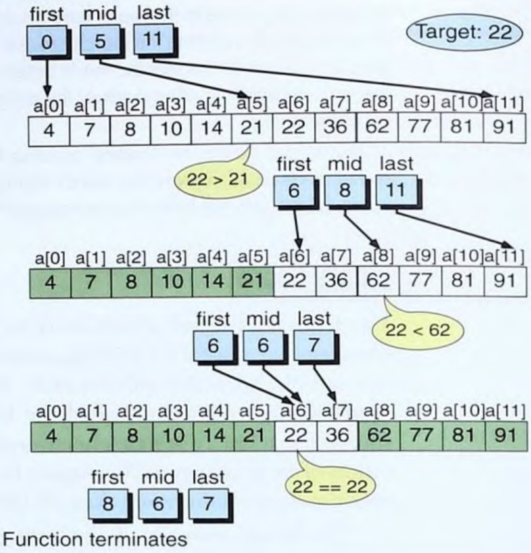


FIGURE 8-31 Binary Search Example

At index location 5, we discover that the target is greater than the list value (22 > 21). We can therefore eliminate the array locations 0 through 5. (Note that mid is automatically eliminated.) To narrow our search, we assign mid + 1 to first and repeat the search.

The next loop calculates mid with the new value for first and determines that the midpoint is now 8.

```
mid = (6 + 11) / 2 = 17 / 2 = 8
```

Again we test the target to the value at mid, and this time we discover that the target is less than the list value (22 < 62). This time we adjust the ends of the list by setting last to mid – 1 and recalculate mid. This eliminates elements 8 through 11 from consideration. We have now arrived at index location 6, whose value matches our target. This stops the search[5]. Figure 8-31 traces the logic we have just described.

Target Not Found

A more interesting case occurs when the target is not in the list. We must construct our search algorithm so that it stops when we have checked all possible locations. This is done in the binary search by testing for first and last crossing; that is, we are done when first becomes greater than last. Thus,

[5] To terminate the search, we set first to last + 1. See Program 8-14.

only two conditions terminate the binary search algorithm: Either the target is found or `first` becomes larger than `last`. For example, imagine we want to find 11 in our binary search array. This situation is shown in Figure 8-32.

In this example, the loop continues to narrow the range as we saw in the successful search, until we are examining the data at index locations 3 and 4. These settings of `first` and `last` set the `mid` index to 3.

```
mid = (3 + 4) / 2 = 7 / 2 = 3
```

The test at index location 3 indicates that the target is greater than the list value, so we set `first` to `mid + 1` or `4`. We now test the data at location 4 and discover that `11` < `14`.

```
mid = (4 + 4) / 2 = 8/ 2 = 4
```

At this point, we have discovered that the target should be between two adjacent values; in other words, it is not in the list. We see this algorithmically because `last` is set to `mid - 1`, which makes `first` greater than `last`, the signal that the value we are looking for is not in the list.

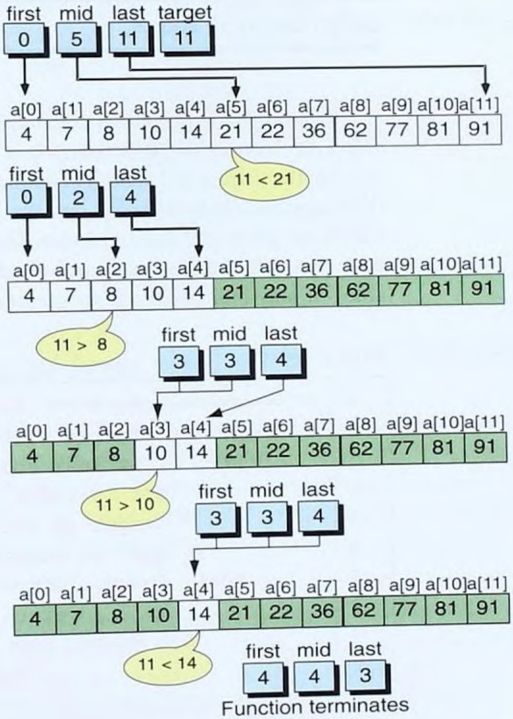


FIGURE 8-32 Unsuccessful Binary Search Example

Once we fully understand the logic, we can design the program. Figure 8-33 contains the design in the form of a flowchart.

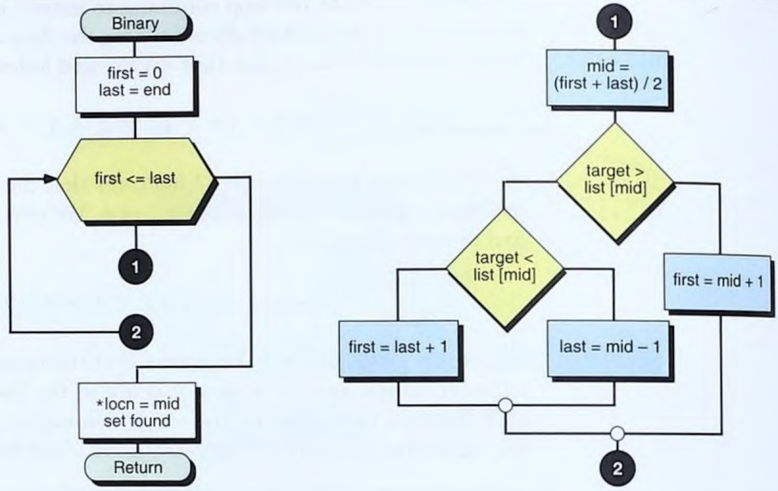


FIGURE 8-33 Design for Binary Search

Program 8-14 contains the implementation of the binary search algorithm we have been describing. It is constructed along the same design we saw for the sequential search. The first three parameters describe the list and the target we are looking for, and the last parameter contains the address into which we place the located index. One point worth noting: When we terminate the loop with a not-found condition, the index returned is unpredictable—it may indicate the node greater than or less than the value in `target`.

PROGRAM 8-14 Binary Search

```
/* =================== binarySearch ===================
   Search an ordered list using Binary Search
      Pre   list must contain at least one element
            end is index to the largest element in list
            target is the value of element being sought
            locn is address for located target index
      Post  Found: locn = index to target element
                   return 1 (found)
          Not Found: locn = element below or above target
                       return 0 (not found)
```

continued

PROGRAM 8-14 Binary Search *(continued)*

```
*/
bool binarySearch (int list[], int  end,
                   int target, int* locn)
{
// Local Declarations
   int first;
   int mid;
   int last;

// Statements
   first = 0;
   last = end;
   while (first <= last)
      {
       mid = (first + last) / 2;
       if (target > list[mid])
          // look in upper half
          first = mid + 1;
       else if (target < list[mid])
          // look in lower half
          last = mid - 1;
       else
          // found equal: force exit
          first = last + 1;
      } // end while
   *locn = mid;
   return target == list [mid];
}  // binarySearch
```

8.7 Two-Dimensional Arrays

The arrays we have discussed so far are known as **one-dimensional arrays** because the data are organized linearly in only one direction. Many applications require that data be stored in more than one dimension. One common example is a **table**, which is an array that consists of rows and columns. Figure 8-34 shows a table, which is commonly called a **two-dimensional array**.

Although a two-dimensional array is exactly what is shown by Figure 8-34, C looks at it in a different way. It looks at the two-dimensional array as an array of arrays. In other words, a two-dimensional array in C is an array of one-dimensional arrays. This concept is shown in Figure 8-35.

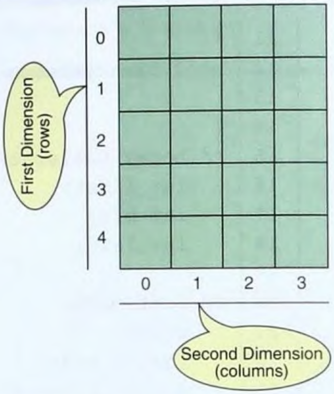


FIGURE 8-34 Two-dimensional Array

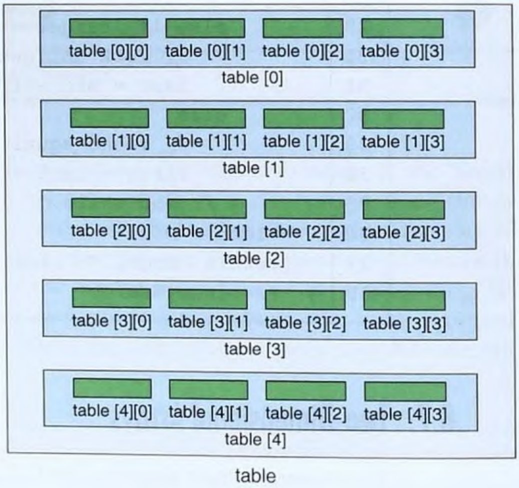


FIGURE 8-35 Array Of Arrays

Declaration

Two-dimensional arrays, like one-dimensional arrays, must be declared before being used. Declaration tells the compiler the name of the array, the type of each element, and the size of each dimension. Two-dimensional arrays can be either fixed length or variable length.

As we saw with the one-dimensional array, the size of a fixed-length array is a constant and must have a value at compilation time. For example, the array shown in Figure 8-35 can be declared and defined as follows:

```
int table[5][4];
```

By convention, the first dimension specifies the number of **rows** in the array. The second dimension specifies the number of **columns** in each row.

The rules for variable-length arrays follow the same concepts. To define a variable-length array, we would use the following statement.

```
int table5x4 [rows][cols];
```

Remember, however, that the dimensions must be set before the declaration or a compiler error occurs.

Initialization

As we noted before, the definition of a fixed-length array only reserves memory for the elements in the array. No values will be stored. If we don't initialize the array, the contents are unpredictable. Generally speaking, fixed-length arrays should be initialized.

Initialization of array elements can be done when the array is defined. As previously noted, variable-length arrays may not be initialized when they are defined. As with one-dimensional arrays, the values must be enclosed in braces. This time, however, there is a set of data for each dimension in the array. So for `table`, we will need 20 values. One way to initialize it is shown below.

```
int table[5][4] =
         { 0,  1,  2,  3, 10, 11, 12, 13, 20, 21,
          22, 23, 30, 31, 32, 33, 40, 41, 42, 43};
```

We highly recommend, however, that nested braces be used to show the exact nature of the array. For example, array `table` is better initialized as shown below.

```
int table [5][4] =
         {
          { 0,  1,  2,  3},
          {10, 11, 12, 13},
          {20, 21, 22, 23},
          {30, 31, 32, 33},
          {40, 41, 42, 43}
         };  // table
```

In this example, we define each row as a one-dimensional array of four elements enclosed in braces. The array of five rows also has its set of braces.

Note that we use commas between the elements in the rows and also commas between the rows.

When we discussed one-dimensional arrays, we said that if the array is completely initialized with supplied values, we do not need to specify the size of the array. This concept carries forward to multidimensional arrays, except that only the first dimension can be omitted. All others must be specified. The format is shown below.

```
int table [][4] =
      {
       { 0,  1,  2,  3},
       {10, 11, 12, 13},
       {20, 21, 22, 23},
       {30, 31, 32, 33},
       {40, 41, 42, 43}
      };  // table
```

To initialize the whole array to zeros, we need only specify the first value, as shown below.

```
int table [5][4] = {0};
```

Inputting Values

Another way to fill up the values is to read them from the keyboard. For a two-dimensional array, this usually requires nested *for* loops. If the array is an *n* by *m* array, the first loop varies the row from zero to *n* – 1. The second loop varies the column from zero to *m* – 1. The code to fill the array in Figure 8-35 is shown below.

```
for (row = 0; row < 5; row++)
    for (column = 0; column < 4; column++)
       scanf ("%d", &table [row][column]);
```

When the program runs, we enter the 20 values for the elements and they are stored in the appropriate locations.

Outputting Values

We can also print the value of the elements one by one, using two nested loops. Again, the first loop controls the printing of the rows and the second loop controls the printing of the columns. To print the table in its table format, a newline is printed at the end of each row. The code to print Figure 8-35 is shown below.

```
for (row = 0; row < 5; row++)
    {
```

continued

```
    for (column = 0; column < 4; column++)
       printf("%8d", table [row][column]);
    printf("\n");
   } // for
```

Accessing Values

Individual elements can be initialized using the assignment operator.

```
table[2][0] = 23;
table[0][1] = table[3][2] + 15;
```

Let's assume that we want to initialize our 5 × 4 array as shown below.

```
00      01      02      03
10      11      12      13
20      21      22      23
30      31      32      33
40      41      42      43
```

One way to do this would be to code the values by hand. However, it is much more interesting to examine the pattern and then assign values to the elements in the array using an algorithm. What pattern do you see? One is that the value in each element increases by one from its predecessor in the row. Another is that the first element in each row is the row index times 10. With these two patterns, we should be able to write nested loops to fill the array. The code to initialize the patterns for the array is shown in Program 8-15.

PROGRAM 8-15 Fill Two-dimensional Array

```
/* ==================== fillArray ====================
   This function fills array such that each array element
   contains a number that, when viewed as a two-digit
   integer, the first digit is the row number and the
   second digit is the column number.
      Pre  table is array in memory
           numRows is number of rows in array
      Post array has been initialized
*/
void fillArray (int table[][MAX_COLS], int numRows)
{
// Statements
   for (int row = 0; row < numRows; row++)
      {
       table [row][0] = row * 10;
```

continued

PROGRAM 8-15 Fill Two-dimensional Array *(continued)*

```
       for (int col = 1; col < MAX_COLS; col++)
           table [row][col] = table [row][col - 1] + 1;
      } // for
   return;
}  // fillArray
```

Memory Layout

As discussed earlier, the indexes in the definition of a two-dimensional array represent rows and columns. This format maps to the way the data are laid out in memory. If we were to consider memory as a row of bytes with the lowest address on the left and the highest address on the right, then an array would be placed in memory with the first element to the left and the last element to the right. Similarly, if the array is a two-dimensional array, then the first dimension is a row of elements that are stored to the left. This is known as "row-major" storage and is seen in Figure 8-36.[6]

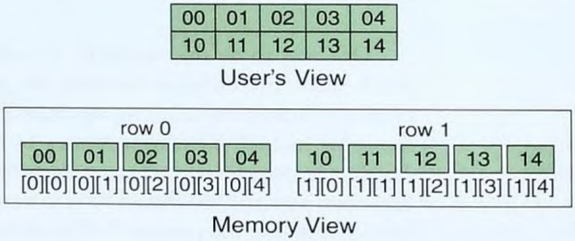


FIGURE 8-36 Memory Layout

Memory Example: Map One Array to Another

To further examine how data are laid out in memory, look at Program 8-16, which converts a two-dimensional array to a one-dimensional array.

PROGRAM 8-16 Convert Table to One-dimensional Array

```
/* This program changes a two-dimensional array to the
   corresponding one-dimensional array.
      Written by:
```

continued

[6] At least one language, FORTRAN, reverses the data values placement in memory. It stores data by columns.

PROGRAM 8-16 Convert Table to One-dimensional Array *(continued)*

```c
      Date:
*/
#include <stdio.h>
#define ROWS 2
#define COLS 5

int main (void)
{
// Local Declarations
   int table [ROWS] [COLS] =
      {
       {00, 01, 02, 03, 04},
       {10, 11, 12, 13, 14}
      }; // table
   int line [ROWS * COLS];

// Statements
   for (int row = 0; row < ROWS; row++)
      for (int column = 0; column < COLS; column++)
         line[row * COLS + column] = table[row][column];

   for (int row = 0; row < ROWS * COLS; row++)
       printf(" %02d ", line[row]);

   return 0;
}  // main
```

```
Results:
00  01  02  03  04  10  11  12  13  14
```

Program 8-16 Analysis In Program 8-16 we use nested *for* loops to make the conversion. The first loop controls the table rows, and the second loop controls the columns within each row. Since we know how many elements are in each row of the two-dimensional array, we can map it to the one-dimensional array by simply multiplying the row index by the number of elements in each row. When we are in row zero, we multiply the number of elements in one row (designed by the defined constant, COLS) by the row number (0) and add the result to the column. This maps the elements in the two-dimensional array to the beginning of the receiving array. When we are in row 1, we add the number of elements in one row times the row (1) to the current column, which maps the elements to the next set (in this case, 5...9). To generalize, we add the product of the row and the number of elements in a row to the column to determine the receiving location.

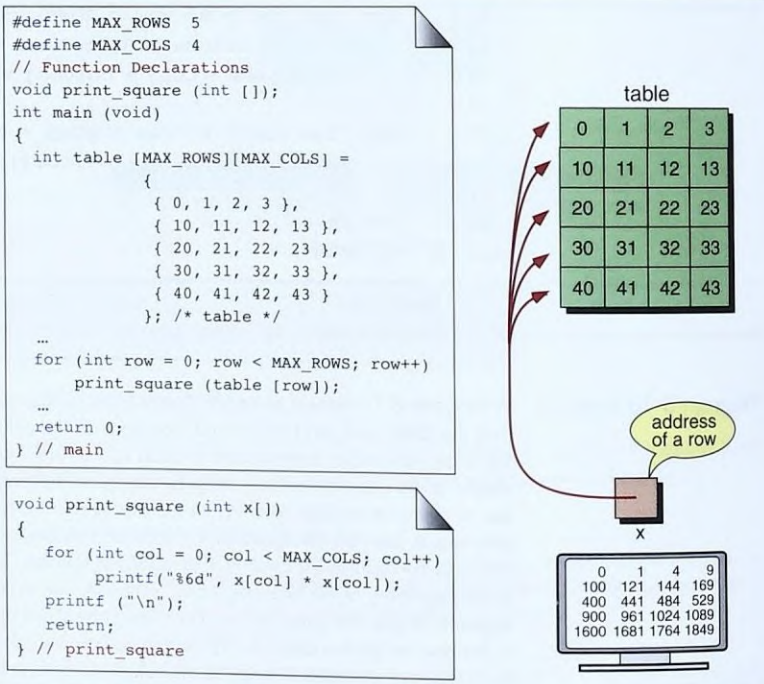
Study the technique we used to define the number of elements in the rows and columns. Note how they are used not only to define the arrays but also to control the loop execution. Copy this program, and then run it with different row–column sizes to help understand the differences. (You will need to change the initialization also. You might want to consider initializing the values by assignments within *for* loops.)

Passing A Two-Dimensional Array

With two-dimensional arrays, we have three choices for passing parts of the array to a function. (1) We can pass individual elements, as shown in "Passing Individual Elements" in Section 8.3. (2) We can pass a row of the array. This is similar to passing an array, as we saw in "Passing the Whole Array" in Section 8.3. (3) Finally, we can pass the whole array.

Passing A Row

Passing a row of the array is rather interesting. We pass a whole row by indexing the array name with only the row number. Referring to Figure 8-37, we see that a row is four integers. When we pass the row, therefore, the receiving function receives a one-dimensional array of four integers. The *for* loop in `print_square` prints the square of each of the four elements. After printing all the values, the function advances to the next line on the console and returns. The *for* loop in *main* calls `print_square` five times so that the final result is a table of the values squared shown on the monitor.



```
#define MAX_ROWS  5
#define MAX_COLS  4
// Function Declarations
void print_square (int []);
int main (void)
{
  int table [MAX_ROWS][MAX_COLS] =
             {
              { 0, 1, 2, 3 },
              { 10, 11, 12, 13 },
              { 20, 21, 22, 23 },
              { 30, 31, 32, 33 },
              { 40, 41, 42, 43 }
             }; /* table */
  ...
  for (int row = 0; row < MAX_ROWS; row++)
      print_square (table [row]);
  ...
  return 0;
} // main
```

```
void print_square (int x[])
{
   for (int col = 0; col < MAX_COLS; col++)
        printf("%6d", x[col] * x[col]);
   printf ("\n");
   return;
} // print_square
```

table

0	1	2	3
10	11	12	13
20	21	22	23
30	31	32	33
40	41	42	43

```
   0    1    4    9
 100  121  144  169
 400  441  484  529
 900  961 1024 1089
1600 1681 1764 1849
```

FIGURE 8-37 Passing a Row

Note that we have declared the array dimensions as defined constants. This allows us to symbolically refer to the limits of the array in both the array definition and the *for* loops. Now, if we need to change the size of the array, all that is necessary is to change the *define* declarations, and any array initializers, and recompile the program.

We could code the print function array in Figure 8-37 as a variable-length array of one dimension. To do so, we would need to change the function definition and the calling statement to pass the array size as shown in the following example.

```
// Function call
print_square (MAX_COLS, table);

// Function definition
void print_square (int size, x[size])
...
} // print_square
```

Passing the Whole Array

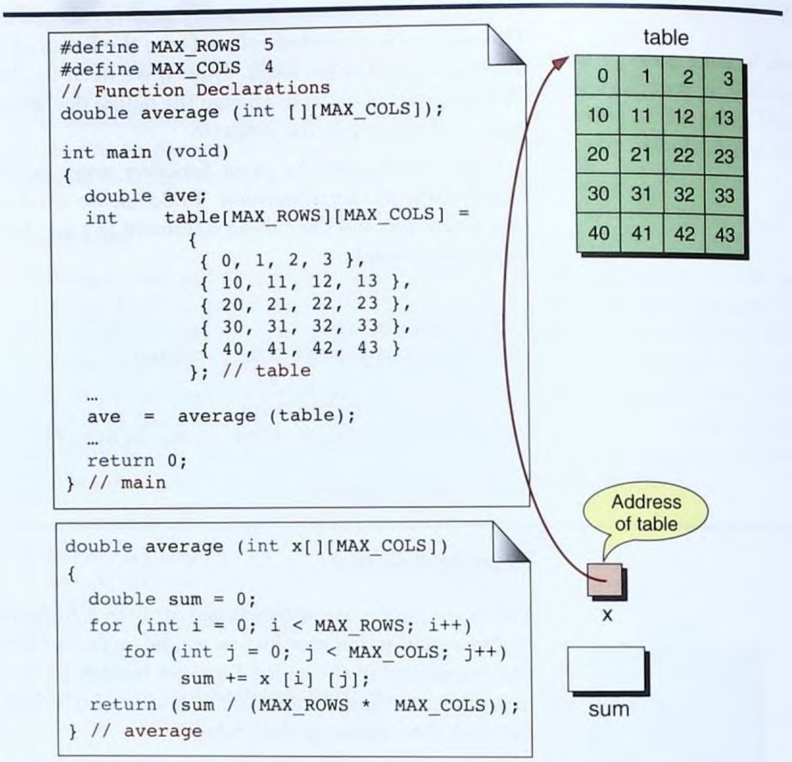
When we pass a two-dimensional array to a function, we use the array name as the actual parameter just as we did with one-dimensional arrays. The formal parameter in the called function header, however, must indicate that the array has two dimensions. This is done by including two sets of brackets, one for each dimension, as shown below.

```
double average (int table[][MAX_COLS])
```

Note that again we do not need to specify the number of rows in a fixed-length array. It is necessary, however, to specify the size of the second dimension. Thus, we specified the number of columns in the second dimension (`MAX_COLS`). In summary, to pass two-dimensional arrays to functions:

1. The function must be called by passing only the array name.
2. In the function definition, the formal parameter is a two-dimensional array, with the size of the *second* dimension required for a fixed-length array.
3. In the function definition for a variable-length array, the size of all dimensions must be specified.

For example, we can use a function to calculate the average of the integers in an array. In this case, we pass the name of the array to the function as seen in Figure 8-38.



```
#define MAX_ROWS  5
#define MAX_COLS  4
// Function Declarations
double average (int [][MAX_COLS]);

int main (void)
{
  double ave;
  int    table[MAX_ROWS][MAX_COLS] =
           {
            { 0, 1, 2, 3 },
            { 10, 11, 12, 13 },
            { 20, 21, 22, 23 },
            { 30, 31, 32, 33 },
            { 40, 41, 42, 43 }
           }; // table
  …
  ave  =  average (table);
  …
  return 0;
} // main
```

```
double average (int x[][MAX_COLS])
{
  double sum = 0;
  for (int i = 0; i < MAX_ROWS; i++)
      for (int j = 0; j < MAX_COLS; j++)
           sum += x [i] [j];
  return (sum / (MAX_ROWS *  MAX_COLS));
} // average
```

0	1	2	3
10	11	12	13
20	21	22	23
30	31	32	33
40	41	42	43

FIGURE 8-38 Calculate Average of Integers in Array

EXAMPLE 8-5 Two-dimensional Array

Write a program that fills the left-to-right diagonal of a square matrix (a two-dimensional array with an equal number of rows and columns) with zeros, the lower left triangle with –1s, and the upper right triangle with +1s. The output of the program, assuming a six-by-six matrix is shown in Figure 8-39.

0	1	1	1	1	1
-1	0	1	1	1	1
-1	-1	0	1	1	1
-1	-1	-1	0	1	1
-1	-1	-1	-1	0	1
-1	-1	-1	-1	-1	0

FIGURE 8-39 Example of Filled Matrix

The program code is shown in Program 8-17.

PROGRAM 8-17 Fill Matrix

```
/* This program fills the diagonal of a matrix (square
   array) with 0, the lower left triangle with -1, and
   the upper right triangle with 1.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int table [6][6];

// Statements
   for (int row = 0; row < 6; row++)
      for (int column = 0; column < 6; column++)
         if (row ==  column)
             table [row][column] = 0;
         else if (row > column)
             table [row][column] = -1;
         else
             table [row][column] = 1;

   for (int row = 0; row < 6; row++)
      {
       for (int column = 0; column < 6; column++)
          printf("%3d", table[row][column]);
       printf("\n");
      } // for row
   return 0;
}  // main
```

Program 8-17 Analysis This is a rather simple pattern problem, similar to many we saw in Chapter 6. The only difference is that now we are creating the pattern in array elements. Since there are two dimensions, we need two loops to control the pattern. If the row equals the column, we assign 0 to the element. If the row is greater than the column, then we are in the lower half of the matrix, so we assign −1. And if the row is less than the column, then we are in the upper half of the matrix and we assign +1.

8.8 Multidimensional Arrays

Multidimensional arrays can have three, four, or more dimensions. Figure 8-40 shows an array of three dimensions. Note the terminology used to describe the array. The first dimension is called a **plane**, which consists of rows and columns. Arrays of four or more dimensions can be created and used, but they are difficult to draw.

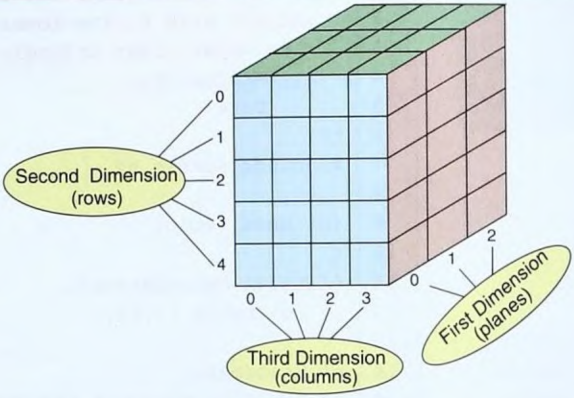


FIGURE 8-40 A Three-dimensional Array (3 x 5 x 4)

Although a three-dimensional array is exactly what is shown in Figure 8-40, the C language looks at it in a different way. It takes the three-dimensional array to be an array of two-dimensional arrays. It considers the two-dimensional array to be an array of one-dimensional arrays. In other words, a three-dimensional array in C is an array of arrays of arrays. This concept also holds true for arrays of more than three dimensions. The C view of a three-dimensional array is seen in Figure 8-41.

Declaring Multidimensional Arrays

Multidimensional arrays, like one-dimensional arrays, must be declared before being used. Declaration tells the compiler the name of the array, the type of each element, and the size of each dimension. The size of the fixed-length array is a constant and must have a value at compilation time. The three-dimensional array seen in Figure 8-41 can be declared as follows for a fixed-length array.

```
int table[3][5][4];
```

The definition for the same table as a variable-length array requires three variables, one for the rows, one for the columns, and one for the planes. Given these variables, it would be defined as shown in the following example.

```
int table[planes][rows][cols];
```

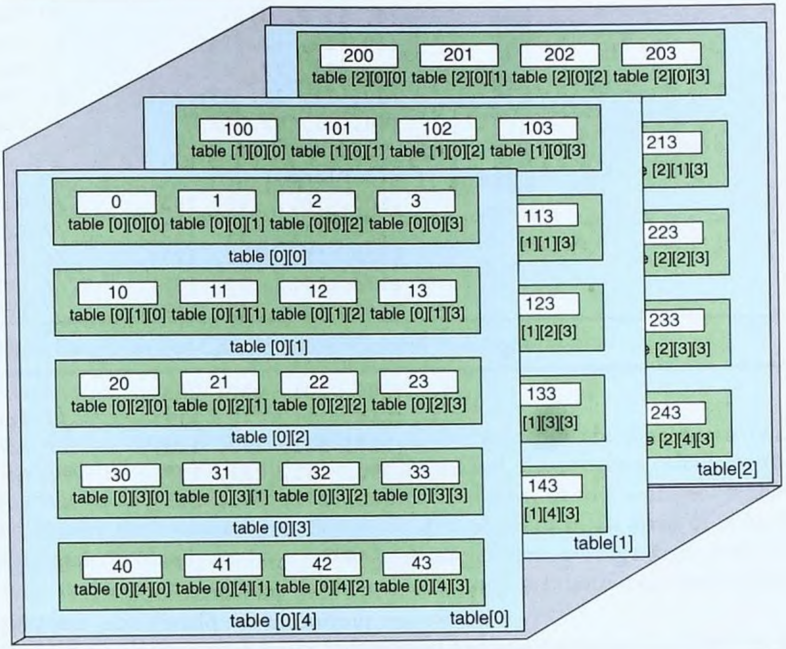


FIGURE 8-41 C View of Three-dimensional Array

Initialization

As we said before, declaration and definition only reserve space for the elements in the array. No values will be stored in the array. If we want to store values, we must either initialize the elements, read values from the keyboard, or assign values to each individual element. Following the rules we discussed for one- and two-dimensional arrays, only fixed-length multidimensional arrays can be initialized when they are defined.

Once more, initialization is simply an extension of the concept we saw for initializing a two-dimensional array (see "Initialization" in Section 8.7). For the three-dimensional array, we nest each plane in a set of brackets. For each plane, we bracket the rows as we did for the two-dimensional array. When we group the data by plane and row as we have done in the following code, the reader will be able to visualize the array with ease. We have added comments to make it even easier to read the values.

```
int table[3][5][4] =
   {
     {  // Plane 0
```

continued

```
        {0, 1, 2, 3},                  // Row 0
        {10, 11, 12, 13},              // Row 1
        {20, 21, 22, 23},              // Row 2
        {30, 31, 32, 33},              // Row 3
        {40, 41, 42, 43}               // Row 4
      },
      {  // Plane 1
        {100, 101, 102, 103},          // Row 0
        {110, 111, 112, 113},          // Row 1
        {120, 121, 122, 123},          // Row 2
        {130, 131, 132, 133},          // Row 3
        {140, 141, 142, 143}           // Row 4
      },
      {  // Plane 2
        {200, 201, 202, 203},          // Row 0
        {210, 211, 212, 213},          // Row 1
        {220, 221, 222, 223},          // Row 2
        {230, 231, 232, 233},          // Row 3
        {240, 241, 242, 243}           // Row 4
      }
    }; // table
```

As we saw previously, the plane's size, and only the plane's size, does not need to be specified when we use explicit initialization. The size of all dimensions after the first must be explicitly stated.

Of course, if we want to initialize all the elements to zero, we can simply initialize only the first element to zero and let the compiler generate the code to initialize the rest of the array.

```
int table [3][5][4] = {0};
```

8.9 Programming Example—Calculate Averages

In Chapters 4 and 5 we introduced the software engineering concept of top-down development and a programming concept known as incremental development. In this chapter, we develop an example that contains many of the programming techniques found in array problems and implement it incrementally. It contains three arrays: a two-dimensional array of integers and two one-dimensional arrays of averages, one for rows and one for columns.

When we work with a large program with many different data structures, in this case three arrays, it helps to draw a picture of the arrays. We can then see how the different arrays work together to solve the problem. A picture of the array structures and their relationships is shown in Figure 8-42.

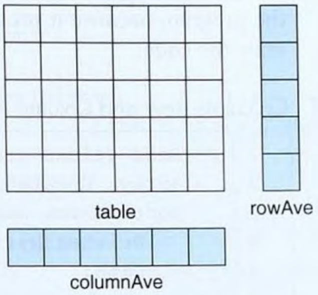


FIGURE 8-42 Data Structures For Calculate Row–Column Averages

The program begins by requesting the user provide data for a two-dimensional array. Once the array has been filled, the program calculates the average of each row and places it in a parallel array of row averages. It then calculates the average for each column and places it in an array of column averages. Although we have represented the column-average array horizontally and the row-average array vertically, they are both one-dimensional arrays.

When all the calculations are complete, the program calls a function to print the array with the row averages at the end of each row and the column averages at the bottom of each column. The structure chart for the program is seen in Figure 8-43.

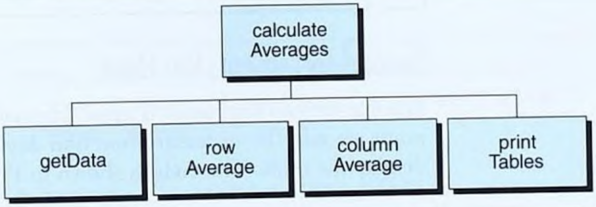


FIGURE 8-43 Calculate Row–Column Average Design

First Increment: *main*

In our previous examples, the first increment included *main* and its first subfunction. Because our data are more complex, we limit it to just *main* in this example. The first increment contains the global include statements we anticipate needing and the known data definitions. In *main* we define our local structures, including the three arrays. We then compile the program and correct any coding errors found by the compiler. Note that we don't run the program because it produces no output at this stage. Program 8-18 contains the code.

PROGRAM 8-18 Calculate Row and Column Averages: *main*

```
/* Read values from keyboard into a two-dimensional
   array. Create two one-dimensional arrays that
   contain row and column averages.
      Written by:
      Date:
*/
#include <stdio.h>

#define MAX_ROWS 5
#define MAX_COLS 6

int main (void)
{
// Local Declarations
   int   table     [MAX_ROWS][MAX_COLS];

   float rowAve    [MAX_ROWS] = {0};
   float columnAve [MAX_COLS] = {0};

// Statements
   return 0;
}  // main
```

Second Increment: Get Data

Having verified that *main* is error free, we are ready to write get data. At this point we add the necessary function declaration and a print loop in *main* to display the table. The code is shown in Program 8-19.

PROGRAM 8-19 Calculate Row and Column Averages: Get Data

```
/* Read values from keyboard into a two-dimensional
   array. Create two one-dimensional arrays that
   contain row and column averages.
      Written by:
      Date:
*/
#include <stdio.h>

#define MAX_ROWS 5
```

continued

PROGRAM 8-19 Calculate Row and Column Averages: Get Data *(continued)*

```
#define MAX_COLS 6

// Function Declaration
void  getData       (int   table[][MAX_COLS]);

int main (void)
{
// Local Declarations
   int   table     [MAX_ROWS][MAX_COLS];

   float rowAve    [MAX_ROWS] = {0};
   float columnAve [MAX_COLS] = {0};

// Statements
   getData        (table);

   printf("\n**Tables built\n");
   for (int i = 0; i < MAX_ROWS; i++)
       {
        for (int y = 0; y < MAX_COLS; y++)
             printf("%4d", table[i][y]);
        printf("\n");
       } // for i

   return 0;
}  // main

/* ===================== getData ====================
   Reads data and fills two-dimensional array.
      Pre   table is empty array to be filled
      Post  array filled
*/
void getData (int table[][MAX_COLS])
{
// Statements
   for (int row = 0; row < MAX_ROWS; row++)
       for (int col = 0; col < MAX_COLS; col++)
           {
            printf("\nEnter integer and <return>: ");
            scanf("%d", &table[row][col]);
           } // for col
   return;
} // getData
// =============== End of Program ================
```

continued

PROGRAM 8-19 Calculate Row and Column Averages: Get Data *(continued)*

```
Results:
**Tables built
  10  12  14  16  18  20
  22  24  26  28  30  23
  25  27  29  31  33  35
  39  41  43  45  47  49
  51  53  55  57  59  61
```

Program 8-19 Analysis After running the program we carefully examine the results to make sure that the new function worked correctly. Because we created our test data before we ran the program, we knew what the results should be. A simple comparison of the test data and the results verifies that the table was built correctly.

Third Increment: Calculate Row Averages

We are now ready to write the code for the averages. We begin with the calculation for the rows. This requires nested *for* loops, the outer loop steps through the rows and the inner loop steps through the columns within a row. The code is shown in Program 8-20.

PROGRAM 8-20 Calculate Row and Column Averages: Row Averages

```
/* Read values from keyboard into a two-dimensional
   array. Create two one-dimensional arrays that
   contain row and column averages.
      Written by:
      Date:
*/
#include <stdio.h>

#define MAX_ROWS 5
#define MAX_COLS 6

// Function Declaration
void getData    (int   table[][MAX_COLS]);
void rowAverage (int   table[][MAX_COLS],
                 float rowAvrg []);

int main (void)
{
// Local Declarations
   int   table     [MAX_ROWS][MAX_COLS];

   float rowAve    [MAX_ROWS] = {0};
```

continued

PROGRAM 8-20 Calculate Row and Column Averages: Row Averages *(continued)*

```
   float columnAve [MAX_COLS] = {0};

// Statements
   getData    (table);
   rowAverage (table, rowAve);

   printf("\n**Tables built\n");
   for (int i = 0; i < MAX_ROWS; i++)
        {
         for (int y = 0; y < MAX_COLS; y++)
              printf("%4d", table[i][y]);
         printf("\n");
        } // for i
   printf("\n");

   printf("**Row averages\n");
   for (int i = 0; i < MAX_ROWS; i++)
        printf("%6.1f", rowAve[i]);
   printf("\n");

   return 0;
}  // main

/* ==================== getData ====================
   Reads data and fills two-dimensional array.
      Pre   table is empty array to be filled
      Post  array filled
*/
void getData (int table[][MAX_COLS])
{
// Statements
   for (int row = 0; row < MAX_ROWS; row++)
      for (int col = 0; col < MAX_COLS; col++)
          {
           printf("\nEnter integer and <return>: ");
           scanf("%d", &table[row][col]);
          } // for col
   return;
} // getData

/* =================== rowAverage ===================
   This function calculates the row averages for a table
      Pre   table has been filled with values
      Post  averages calculated and in average array
```

continued

PROGRAM 8-20 Calculate Row and Column Averages: Row Averages *(continued)*

```
*/
void rowAverage (int    table[][MAX_COLS],
                 float  rowAvrg [])
{
// Statements
   for (int row = 0; row < MAX_ROWS; row++)
       {
        for (int col = 0; col < MAX_COLS; col++)
            rowAvrg[row] += table [row][col];
        rowAvrg [row] /=  MAX_COLS;
       } // for row
   return;
}  // rowAverage
// =============== End of Program ================
```

```
Results:
**Tables built
  10  12  14  16  18  20
  22  24  26  28  30  23
  25  27  29  31  33  35
  39  41  43  45  47  49
  51  53  55  57  59  61

**Row averages
  15.0  25.5  30.0  44.0  56.0
```

Fourth Increment: Calculate Column Averages

The calculation of the column averages is very similar to the calculation of the row averages. We leave its incremental development for a problem at the end of the chapter.

Fifth Increment: Print Tables

The final increment completes the program by printing the three tables. Note that we present the results with the row averages at the end of the rows and the column averages at the bottom of the columns. This is a natural and logical presentation from a user's perspective. The code for the fifth increment is shown in Program 8-21. Note that the column totals will be displayed as zero until the fourth increment is completed.

PROGRAM 8-21 Calculate Row and Column Averages: Print Tables

```
/* Read values from keyboard into a two-dimensional
   array. Create two one-dimensional arrays that
```

continued

PROGRAM 8-21 Calculate Row and Column Averages: Print Tables *(continued)*

```
     contain row and column averages.
        Written by:
        Date:
*/
#include <stdio.h>

#define MAX_ROWS 5
#define MAX_COLS 6

// Function Declaration
void getData        (int   table[][MAX_COLS]);
void rowAverage     (int   table[][MAX_COLS],
                     float rowAvrg []);
void colAverage     (int   table[][MAX_COLS],
                     float colAvrg []);
void  printTables   (int   table[][MAX_COLS],
                     float rowAvrg[],
                     float colAvrg[]);

int main (void)
{
// Local Declarations
   int   table  [MAX_ROWS][MAX_COLS];

   float rowAve [MAX_ROWS] = {0};
   float colAve [MAX_COLS] = {0};

// Statements
   getData (table);
   rowAverage (table, rowAve);
// colAverage (table, colAve);

   printf("\n");
   printTables (table, rowAve, colAve);
   return 0;
}  // main

/* ==================== getData ====================
   Reads data and fills two-dimensional array.
      Pre   table is empty array to be filled
      Post  array filled
*/
void getData (int table[][MAX_COLS])
{
```

continued

PROGRAM 8-21 Calculate Row and Column Averages: Print Tables *(continued)*

```
// Statements
   for (int row = 0; row < MAX_ROWS; row++)
      for (int col = 0; col < MAX_COLS; col++)
          {
           printf("\nEnter integer and <return>: ");
           scanf("%d", &table[row][col]);
          } // for col
   return;
} // getData

/* ================== rowAverage ===================
   This function calculates the row averages for a table
      Pre   table has been filled with values
      Post  averages calculated and in average array
*/
void rowAverage (int    table[][MAX_COLS],
                 float  rowAvrg [])
{
// Statements
   for (int row = 0; row < MAX_ROWS; row++)
       {
        for (int col = 0; col < MAX_COLS; col++)
            rowAvrg[row] += table [row][col];
        rowAvrg [row] /=  MAX_COLS;
       } // for row
   return;
}  // rowAverage

/* ================== printTables ===================
   Print data table, with row average at end of each
   row and average of columns below each column.
      Pre   each table filled with its data
      Post  tables printed
*/
void printTables (int    table[][MAX_COLS],
                  float  rowAvrg[],
                  float  colAvrg[])
{
// Statements
   for (int row = 0; row < MAX_ROWS; row++)
       {
        for (int col = 0; col < MAX_COLS; col++)
            printf("%6d", table[row][col]);
        printf("   | %6.2f\n", rowAvrg [row]);
```

continued

PROGRAM 8-21 Calculate Row and Column Averages: Print Tables *(continued)*

```
      } // for row

   printf("----------------------------------------\n");
   printf("    ");
   for (int col = 0; col < MAX_COLS; col++)
        printf("%6.2f", colAvrg[col]);
   return;
}  // printTables
// ================ End of Program =================
```

```
Results:
    10    12    14    16    18    20   |  15.00
    22    24    26    28    30    23   |  25.50
    25    27    29    31    33    35   |  30.00
    39    41    43    45    47    49   |  44.00
    51    53    55    57    59    61   |  56.00
---------------------------------------
      0.00  0.00  0.00  0.00  0.00  0.00
```

8.10 Software Engineering

In this section, we discuss two basic concepts: testing and algorithm efficiency. To be effective, testing must be clearly thought out. We provide some concepts for testing array algorithms by studying sorting and searching. We then continue the algorithm efficiency discussion started in Chapter 6 by studying sort and search algorithms as case studies.

Testing Sorts

As our programs become more complex, we need to spend more time creating test data that will completely validate them. In this section, we examine some techniques for testing sorts.

In general, we should conduct four tests: (1) sort a list of random values, (2) sort a list that is already in sequence, (3) sort a list that is in reverse order, (4) sort a nearly ordered list, such as one in which every tenth item is one position out of sequence. Table 8-2 contains a summary of the tests that we should conduct and some sample test data to show the points.

Test Case	Sample Data
Random data	`5 23 7 78 22 6 19 33 51 11 93 31`
Nearly ordered	`5 6 7 21 19 22 23 31 29 33 51 93`
Ordered — ascending	`5 6 7 11 19 22 23 31 33 51 78 93`
Ordered — descending	`93 78 51 33 31 23 22 19 11 7 6 5`

TABLE 8-2 Recommended Sort Test Cases

Testing Searches

When we test the sequential search, we need only four tests. Three deal with finding an element in the table—find the first, last, and any element in the middle. The last case deals with trying to find an element that is not in the list; look for any value that is not in the middle of the list. Whenever testing an array, always be sure that the test data access the first and last elements of the array. Many array processing errors occur at these locations.

When testing an array search, always access the first and last elements.

The binary search requires the four tests discussed above, plus three more. Since it uses an ordered list, we should try to find a target lower than the first element in the list and another target greater than the last element. Finally, we should find two elements that are in adjacent array locations, such as `list[0]` and `list[1]`. The reason for this test is that the binary search includes logic that divides by two. An error could result in being able to find only even- or only odd-numbered elements. Testing for adjacent locations ensures that such an error won't happen. These test cases are seen in Table 8-3.

Expected Results	Index	Test	Search
found	0	target == list[0]	all
found	1	target == list[1]	binary only
found	*n* – 1	target == list[*n*–1]	all
found	0 < i < *n*	target == list[i]	all
not found	0	target < list[0]	binary only
not found	*n* – 1	target > list[n]	binary only
not found	0 < i < *n*	target != list[i]	all

TABLE 8-3 Test cases for searches

Analyzing Sort Algorithms

We have developed three sort algorithms. We examine each of them in this section.

Bubble Sort Analysis

The bubble sort essentially contains the block of code shown below. Note that "exchange" in the code could be a function call or the three statements shown in Program 8-10.

```
for(current = 0; current < last; current++)
   for (walker = last; walker > current; walker--)
       if (list[walker] < list[walker - 1])
           exchange (walker, walker - 1);
```

To determine the relative efficiency, we need to analyze the two *for* statements. The first *for,* the outer loop, examines each entry in the sort array. It will therefore loop $n - 1$ times.

The inner loop starts at the end of the array and works its way toward the current node as established by the outer loop. The first time it is called, it examines $n - 1$ elements; the second time, $n - 2$ elements, and so forth until it examines only one element. The average number of elements examined, therefore, is determined as shown below.

$$(\mathbf{n}-1)+(\mathbf{n}-2)+\ldots+2+1 = \mathbf{n}\frac{(\mathbf{n}-1)}{2}$$

This is the nested dependent loop. Simplifying the previous formula we get

$$n\left(\frac{n-1}{2}\right) = \frac{1}{2}(n^2 - n)$$

Discarding the coefficient and selecting the larger factor, we see that the dominant factor in the bubble sort is n^2, which in big-O notation would be stated as $O(n^2)$.

The efficiency of the bubble sort is $O(n^2)$.

Selection Sort Analysis

Now let's examine the efficiency of the selection sort shown in Program 8-9. Again, its pivotal logic is essentially shown in the next example.

```
for (current = 0; current < last; current++)
   {
    for (walker = current + 1; walker <= last; walker++)
       if (list[walker] < list[smallest])
           smallest = walker;
   } // for current
```

This algorithm bears a strong resemblance to the bubble sort algorithm we discussed previously. Its first loop looks at every element in the array from the first element (current = 0) to the one just before the last. The inner loop moves from the current element, as determined by `walker`, to the end of the list. This is similar to the bubble sort, except that it works from the lower portion of the array toward the end. Using the same analysis, we see that it will test $n(n - 1) / 2$ elements, which means that the selection sort is also $O(n^2)$.

The efficiency of the selection sort is $O(n^2)$.

Insertion Sort Analysis

The last sort we covered was the insertion sort, Program 8-11. The nucleus of its logic is shown as follows.

```
for (current = 1; current <= last; current++)
    for (walker = current - 1;
         walker >= 0 && !located;)
       if (temp < list[walker])
```

continued

```
          {
           list[walker + 1] = list[walker];
           walker--;
          } // if
```

Does the pattern look familiar? It should. Again we have the same basic nested *for* loop logic that we saw in the bubble sort and the selection sort. The outer loop is executed *n* times, and the inner loop is executed $(n - 1) / 2$ times, giving us $O(n^2)$.

The efficiency of the insertion sort is $O(n^2)$.

As we have demonstrated, all three of these sorts are $O(n^2)$, which means that they should be used only for small lists or lists that are nearly ordered. You will eventually study sorts that are $O(n\log n)$, which is much more efficient for large lists.

Analyzing Search Algorithms

All of the sort algorithms involved nested loops. We now turn our attention to two algorithms that have only one loop, the sequential search and the binary search. Recall that a search is used when we need to find something in an array or other list structure. The *target* is a value obtained from some external source.

Sequential Search Analysis

The basic loop for the sequential search is shown below.

```
while (looker < last && target != list[looker])
       looker++;
```

This is a classic example of a linear algorithm. In fact, in some of the literature, this search is known as a **linear search**. Since the algorithm is linear, its efficiency is $O(n)$.

The efficiency of the sequential search is $O(n)$.

Binary Search Analysis

The binary search locates an item by repeatedly dividing the list in half. Its loop is

```
while (first <= last)
    {
     mid = (first + last) / 2;
```

continued

```
   if (target > list[mid])
      first = mid + 1;
   else if (target < list[mid])
      last = mid - 1;
   else
      first = last + 1;
  } // while
```

This is obviously a loop that divides, and it is therefore a logarithmic loop. This makes the efficiency O(log*n*), which you should recognize as one of the more efficient of all the measures.

The efficiency of the binary search is *O*(log*n*).

Comparing the sequential search and binary search, we see that, disregarding the time required to order the list, the binary search is obviously better for a list of any significant size (see Table 8-4). We therefore recommend the binary search for all but the smallest of lists—that is, lists with less than 50 elements.

Size	Binary	Sequential (Average)	Sequential (Worst Case)
16	4	8	16
50	6	25	50
256	8	128	256
1,000	10	500	1,000
10,000	14	5,000	10,000
100,000	17	50,000	100,000
1,000,000	20	500,000	1,000,000

TABLE 8-4 Comparison of binary and sequential searches

The big-O concept is generally interested only in the largest factor. This tends to significantly distort the efficiency of the sequential sort, in that it is always the worst case. If the search is always successful, it turns out that the efficiency of the sequential search is 1/2*n*. We include the average in Program 8-3 for comparison. (The average for the binary search is only one less than the maximum, so it is less interesting.)

8.11 Tips and Common Programming Errors

1. In an array declared as `array[n]`, the index goes from 0 (not 1) to $n - 1$.
2. Three things are needed to declare and define an array: its name, type, and size.
3. The elements of arrays are not initialized automatically. You must initialize them if you want them to start with known values.
4. To initialize all elements in an array to zero, all you need to do is initialize the first element to zero.
5. To exchange the value of two elements in an array, you need a temporary variable.
6. You cannot copy all elements of one array into another with an assignment statement. You need to use a loop.
7. To pass the whole array to a function, you only use the name of the array as an actual parameter.
8. The most common logic error associated with arrays is an invalid index. An invalid index used with an assignment operator either causes the program to fail immediately or destroys data or code in another part of the program and causes it to fail later.
9. Invalid indexes are often created by invalid coding in a *for* statement. For example, given an array of ten elements, the following *for* statement logic error results in an index value of ten being used. Although it loops 10 times, the indexes are 1 through 10, not zero through 9.

   ```
   for (i = 1; i <= 10; i++)
   ```

10. Another cause of invalid indexes is an uninitialized index. Make sure your indexes are always properly initialized.
11. When initializing an array when it is defined, it is a compile error to provide more initializers than there are elements.
12. It is a compile error to leave out the index operators in an assignment statement, as shown below.

    ```
    float costAry[20];
    ...
    costAry = quantity * price;
    ```

13. It is most likely a logic error to leave out the index operators in a *scanf* statement, as shown in the following example. In this case, `costAry` is the address of the array and the input will be placed in the first element of the array. Even when this is the desired result, you should code it with the index operators as shown.

```
float costAry[20];
…
scanf("%f", costAry);                    // Poor Style
scanf("%f", &costAry[0]);                // Clear code
```

14. It is a compile error to omit the array size in the parameter declaration for any array dimension other than the first.

15. To pass a variable-size array to a function, you need either to include the variable defining the array size or use an asterisk.

8.12 Key Terms

array
binary search
bubble sort
column
exchanging
fixed-length array
frequency array
histogram
index
invalid index
insertion sort
linear search
multidimensional array
one-dimensional array
passing a row
pass the whole array
pass individual elements
plane
reusable code
row
searching
selection sort
sequential search
sort pass
subscript
table
two-dimensional array
variable-length array

8.13 Summary

- A one-dimensional array is a fixed sequence of elements of the same type.
- We use indexes in C to show the position of the elements in an array.
- An array must be declared and defined before being used. Declaration and definition tell the compiler the name of the array, the type of each element, and the size of the array.
- Initialization of all elements of a fixed length array can be done at the time of the declaration and definition.
- If a one-dimensional array is completely initialized when it is declared, it is not necessary to specify the size, but it is recommended.
- When an array is partially initialized, the rest of the elements are set to zero.

- ❑ We can fill the elements of an array by using a loop to read the values from the keyboard or a file.
- ❑ We can access the individual elements of an array using the array name and the index.
- ❑ We can read or write the values of an array using a loop.
- ❑ An array reference is a postfix expression with the opening and closing brackets as operators.
- ❑ C does not do boundary checking on the elements of an array.
- ❑ We can pass an individual element of an array to a function either by value or by address.
- ❑ We can also pass the whole array to a function. In this case, only the address of the array will be passed. When this happens, the function has access to all elements in the array.
- ❑ A frequency array is an array whose elements show the number of occurrences of data values in another array.
- ❑ A histogram is a pictorial representation of a frequency array.
- ❑ A two-dimensional array is a representation of a table with rows and columns.
- ❑ We can pass either a single element, a row, or the whole two-dimensional array to a function.
- ❑ A multidimensional array is an extension of a two-dimensional array to three, four, or more dimensions.
- ❑ An array can be sorted using a sorting algorithm.
- ❑ The selection sort divides the array into sorted and unsorted sublists. In each pass, the algorithm chooses the smallest element from the unsorted sublist and swaps it with the element at the beginning of the unsorted sublist.
- ❑ The bubble sort divides the array into sorted and unsorted sublists. In each pass, the algorithm bubbles the smallest element from the unsorted list into the sorted sublist.
- ❑ The insertion sort divides the array into sorted and unsorted sublists. In each pass, the algorithm inserts the first element from the unsorted list into the appropriate place in the sorted sublist.
- ❑ Searching is the process of finding the location of a target among a list of objects.
- ❑ The sequential search starts at the beginning of the array and searches until it finds the data or hits the end of the list. The data may be ordered or unordered.

- ❑ A binary search is a much faster searching algorithm. In the binary search, each test removes half of the list from further analysis. The data must be ordered.
- ❑ To pass a whole variable-length array to a function, you need to also pass its size.

8.14 Practice Sets

Review Questions

1. The type of all elements in an array must be the same.

   a. True
   b. False

2. Any expression that evaluates to an integral value may be used as an index.

   a. True
   b. False

3. When an array is defined, C automatically sets the value of its elements to zero.

   a. True
   b. False

4. When an array is passed to a function, C passes the value for each element.

   a. True
   b. False

5. Because of its efficiency, the binary search is the best search for any array, regardless of its size and order.

   a. True
   b. False

6. The selection, insertion, and bubble sort are all $O(n^2)$ sorts.

   a. True
   b. False

7. A(n) ________________ is an integral value used to access an element in an array.

   a. constant
   b. element
   c. index
   d. number
   e. variable

8. Which of the following array initialization statements is valid?

   a. `int ary{ } = {1, 2, 3, 4};`
   b. `int ary[ ] = [1, 2, 3, 4];`
   c. `int ary[ ] = {1, 2, 3, 4};`
   d. `int ary{4} = [1, 2, 3, 4];`
   e. `int ary[4] = [1, 2, 3, 4];`

9. Which of the following statements assigns the value stored in *x* to the first element on an array, `ary`?

   a. `ary    = x;`
   b. `ary    = x[0];`
   c. `ary    = x[1];`
   d. `ary[0] = x;`
   e. `ary[1] = x;`

10. Which of the following statements concerning passing array elements is true?

    a. Arrays cannot be passed to functions, because their structure is too complex.
    b. It is not possible to pass just a row of a two-dimensional array to a function.
    c. Only the size of the first dimension is needed when a two-dimensional array is declared in a parameter list.
    d. When an array is passed to a function, it is always passed by reference (only its address is passed).
    e. When a two-dimensional array is passed to a function, the size of the second dimension must be passed as a value parameter.

11. The process through which data are arranged according to their values is known as

    a. arranging
    b. listing
    c. parsing
    d. searching
    e. sorting

12. The ______________ sort finds the smallest element from the unsorted sublist and swaps it with the element at the beginning of the unsorted data.

    a. bubble
    b. exchange
    c. insertion
    d. quick
    e. selection

13. The ______________ search locates the target item by starting at the beginning and moving toward the end of the list.

    a. ascending
    b. binary
    c. bubble
    d. selection
    e. sequential

14. Which of the following statements about a sequential search is false?

    a. Any array can be searched using the sequential search.
    b. If the target is not found, every element in the list is tested.
    c. The efficiency of the sequential search is $O(n)$.
    d. The list must be ordered.
    e. The sequential search is generally recommended only for small lists.

15. Which of the following statements about two-dimensional arrays is true?

    a. A two-dimensional array can be thought of as an array of one-dimensional arrays.
    b. Only the size of the second dimension needs to be declared when the array is used as a parameter.
    c. Two different types can be stored in a two-dimensional array.
    d. The first dimension is known as the column dimension.
    e. When passed to a function, the size of the second dimension must be passed as a value parameter.

Exercises

16. What would be printed by the following program?

```
#include <stdio.h>
int main (void)
{
// Local Declarations
   int list [10] = {0};
// Statements
   for (int i = 0; i < 5;  i++)
       list [2 * i + 1] = i + 2;
   for (int i = 0; i < 10;  i++)
       printf("%d\n", list [i]);
   return 0;
}  // main
```

17. What would be printed by the following program?

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

continued

```
// Local Declarations
   int list [10] = {2, 1, 2, 1, 1, 2, 3, 2, 1, 2};
// Statements
   printf("%d\n", list [2]);
   printf("%d\n", list [list [2]]);
   printf("%d\n", list [list [2] + list [3]]);
   printf("%d\n", list [list [list [2]]]);
   return 0;
}  // main
```

18. What would be printed by the following program?

```
#include <stdio.h>
int main (void)
{
// Local Declarations
   int list [10] = {2, 1, 2, 4, 1, 2, 0, 2, 1, 2};
   int line [10];
// Statements
   for (int i = 0; i < 10; i ++)
       line [i] = list [9 - i];
   for (int i = 0; i < 10; i++)
       printf("%d %d\n", list [i], line [i]);
   return 0;
}  // main
```

19. An array contains the elements shown below. The first two elements have been sorted using a selection sort. What would be the value of the elements in the array after three more passes of the selection sort algorithm?

```
7  8  26  44  13  23  98  57
```

20. An array contains the elements shown below. The first two elements have been sorted using a bubble sort. What would be the value of the elements in the array after three more passes of the bubble sort algorithm? Use the version of bubble sort that starts from the end and bubbles the smallest element.

```
7  8  26  44  13  23  57  98
```

21. An array contains the elements shown below. The first two elements have been sorted using an insertion sort. What would be the value of the elements in the array after three more passes of the insertion sort algorithm?

```
3  13  7  26  44  23  98  57
```

22. We have the following array:

```
47  3  21  32  56  92
```

After two passes of a sorting algorithm, the array has been rearranged as shown below.

```
3  21  47  32  56  92
```

Which sorting algorithm is being used (selection, bubble, insertion)? Defend your answer.

23. We have the following array:

```
80  72  66  44  21  33
```

After two passes of a sorting algorithm, the array has been rearranged as shown below.

```
21  33  80  72  66  44
```

Which sorting algorithm is being used (selection, bubble, insertion)? Defend your answer.

24. We have the following array:

```
47  3  66  32  56  92
```

After two passes of a sorting algorithm, the array has been rearranged as shown below.

```
3  47  66  32  56  92
```

Which sorting algorithm is being used (selection, bubble, insertion)? Defend your answer.

25. An array contains the elements shown below. Using the binary search algorithm, trace the steps followed to find 88. At each loop iteration, including the last, show the contents of `first`, `last`, and `mid`.

```
8  13  17  26  44  56  88  97
```

26. An array contains the elements shown below. Using the binary search algorithm, trace the steps followed to find 20. At each loop iteration, including the last, show the contents of `first`, `last`, and `mid`.

```
8  13  17  26  44  56  88  97
```

27. Both the selection and bubble sorts exchange elements. The insertion sort does not. Explain how the insertion sort rearranges the data without exchanges.

Problems

28. We have two arrays `A` and `B`, each of 10 integers. Write a function that tests if every element of array `A` is equal to its corresponding element in array `B`. In other words, the function must check if `A[0]` is equal to `B[0]`, `A[1]` is equal to `B[1]`, and so forth. The function is to return true if all elements are equal and false if at least one element is not equal.

29. Write a function that reverses the elements of an array so that the last element becomes the first, the second from the last becomes the second, and so forth. The function is to reverse the elements in place—that is, without using another array. (It is permissible to use a variable to hold an element temporarily.) Then write a test driver to test your function. Test it twice, once with an even number of elements in the array and once with an odd number of elements in the array.

30. The Pascal triangle can be used to compute the coefficients of the terms in the expansion of $(a + b)^n$. Write a function that creates a two-dimensional matrix representing the Pascal triangle. In a Pascal triangle, each element is the sum of the element directly above it and the element to the left of the element directly above it (if any). A Pascal triangle of size 7 is shown below.

```
1
1       1
1       2       1
1       3       3       1
1       4       6       4       1
1       5      10      10       5       1
1       6      15      20      15       6       1
```

In the above example, the first element of row[0] and the first two elements of row[1] are set to 1. Then each of the following rows, up to the maximum size, are set with a loop as shown in the following pseudocode:

```
Algorithm PascalTriangle
 1 pascal[0][0] = 1
 2 pascal[1][0] = 1
 3 pascal[1][1] = 1
 4 prevRow      = 1
 5 currRow      = 2
 6 loop (currRow <= size)
    1  pascal[row][0] = 1
    2  col = 1
```

continued

```
         3  loop (col <= currRow)
            1  pascal[ row][ col] =
                  pascal[row - 1][col - 1]
                  + pascal[row - 1][col]
            2  col = col + 1
         4  end loop
      7 end loop
   end PascalTriangle
```

Your program must be able to create the triangle of any size.

31. An international standard book number (ISBN) is used to uniquely identify a book. It is made of 10 digits, as shown in Figure 8-44. Write a function that tests an ISBN to see if it is valid. For an ISBN number to be valid, the weighted sum of the 10 digits must be evenly divisible by 11. The tenth digit may be *x*, which indicates 10.

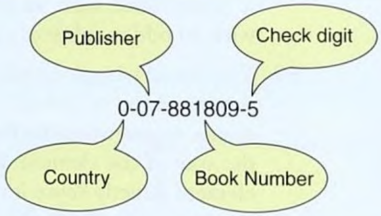


FIGURE 8-44 ISBN for Problem 31

To determine the weighted sum, the value of each position is multiplied by its relative position, starting from the right, and the sum of the products is determined. The calculation of the weighted sum for the ISBN shown above is demonstrated in Table 8-5.

Code	Weight	Weighted Value
0	10	0
0	9	0
7	8	56
8	7	56
8	6	48
1	5	5

continued

TABLE 8-5 Demonstration of ISBN Calculation

Code	Weight	Weighted Value
8	4	32
0	3	0
9	2	18
5	1	5
	Weighted Sum	220

TABLE 8-5 Demonstration of ISBN Calculation *(continued)*

Since the weighted sum modulus 11 is 0, the ISBN number is valid. Test your function with the above example, the ISBN number for this text (see the copyright page), and 0-08-781809-5 (an invalid ISBN—the third and fourth digits are reversed).

32. Another technique to validate the ISBN in Problem 31 is to calculate the sum of the sums modulus 11. The sum of the sums is calculated by adding each digit to the sum of the previous digits, as shown in Table 8-6.

Code	Sum of Digits	Sum of Sums
0	0	0
0	0	0
7	7	7
8	15	22
8	23	45
1	24	69
8	32	101
0	32	133
9	41	174
5	46	220

TABLE 8-6 ISBN Calculation: Sum of Sums Method

Rewrite the function to use the sum-of-the-sums method.

33. Write a function that copies a one-dimensional array of *n* elements into a two-dimensional array of *k* rows and *j* columns. The rows and columns must be a valid factor of the number of elements in the one-dimensional array; that is, $k * j = n$.

34. Write a program that creates an array of 100 random integers in the range 1 to 200 (see Chapter 4 for a discussion of random numbers) and then, using the sequential search, searches the array 100 times using randomly generated targets in the same range. At the end of the program, display the following statistics:

    a. The number of searches completed.

    b. The number of successful searches.

    c. The percentage of successful searches.

    d. The average number of tests per search.

    To determine the average tests per search, you will need to count the number of tests for each search. Hint: Use the comma operator to count the compares.

35. Repeat Problem 34 using the binary search.

36. The sequential search assumes that a list is unordered. If it is used when the list is in fact ordered, the search can be terminated with the target not found whenever the target is less than the current element. Modify Program 8-13 to incorporate this logic.

37. Repeat Problem 34 with the modified search you created in Problem 36.

38. The binary search is a good algorithm to implement using recursion. Rewrite the binary search function using recursion. Then repeat Problem 35 using the recursive function.

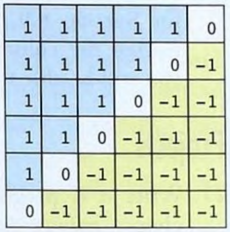
39. Modify the bubble sort to stop as soon as the list is sorted. (See discussion in the analysis of Program 8-10.)

40. Modify the selection sort function to count the number of exchanges needed to order an array of 50 random numbers (see Chapter 4 for a discussion of random numbers). Display the array before and after the sort. At the end of the program, display the total exchanges needed to sort the array.

41. Repeat Problem 40 using the bubble sort.

42. Program 8-7 builds a frequency array and its histogram. If there is an invalid data point in the input, it is displayed and ignored. Modify the program to make the frequency array one element larger than the data range, and use the last element of the array as a count of numbers not in the specified range.

43. Write a program that fills the right-to-left diagonal of a square matrix with zeros, the lower-right triangle with –1s, and the upper-left triangle with +1s. The output of the program, assuming a six-by-six matrix, is shown in Figure 8-45.



1	1	1	1	1	0
1	1	1	1	0	-1
1	1	1	0	-1	-1
1	1	0	-1	-1	-1
1	0	-1	-1	-1	-1
0	-1	-1	-1	-1	-1

FIGURE 8-45 Matrix for Problem 43

44. Given an array of 100 random numbers in the range 1…999, write a function for each of the following processes. In building the array, if the random number is evenly divided by 3 or 7, store it as a negative number.

a. Print the array ten values to a line. Make sure that the values are aligned in rows.
b. Print the odd values, ten to a line.
c. Print the values at the odd numbered index locations, ten to a line.
d. Return a count of the number of even values.
e. Return the sum of all values in the array.
f. Return the location of the smallest value in the array.
g. Return the location of the largest value in the array.
h. Copy all positive values to a new array. Then use process "a" above to print the new array.
i. Copy all negative values to a new array. Then use process "a" above to print the new array.

45. Modify the sort of your choice to sort two parallel arrays. Parallel arrays are two or more arrays that contain related data. For example, one array might contain identification numbers, and a related array might contain corresponding scores in a tournament. The sort will sort only the array containing the identifiers; however, as each element in the sorted array is moved, its corresponding element in the data array must also be moved. After the arrays have been sorted, print out the data as shown below.

```
ID     Score
 1      72
 2      89
 3      98
 4     105
```

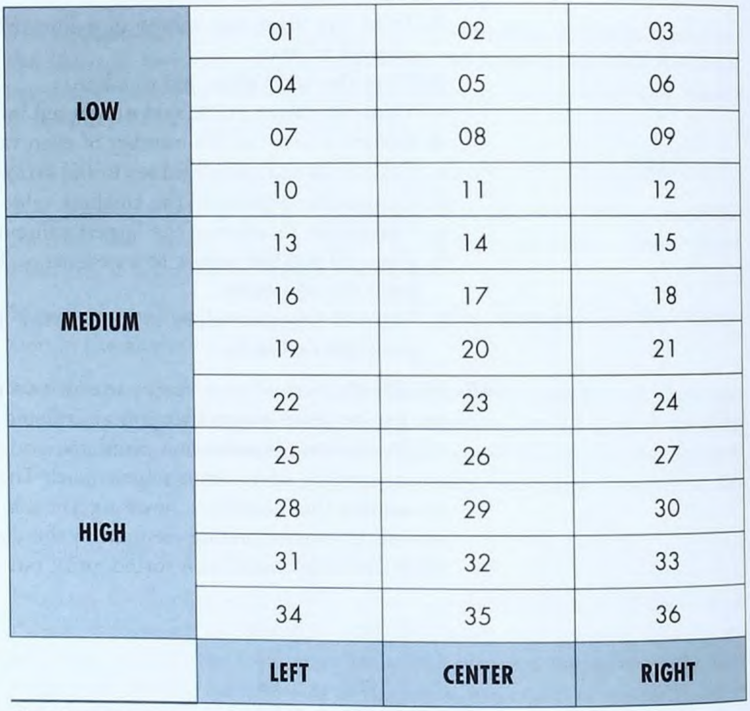
Write a test driver to test your sort. Use the following data—the first value is an identifier and the second value is a score: {18, 90}, {237, 47}, {35, 105}, {5,25}, {76, 739}, {103, 26}, {189, 38}, {22, 110}, {156, 31}, {49, 245}.

46. Write the incremental implementation to calculate the column averages for Section 8.9, "Programming Example—Calculate Averages." Before testing the column average increment, determine the averages so that you will be able to verify that the correct values were calculated.

Projects

47. Write a C program that simulates a guessing game. Each turn, you choose among nine possible guesses. As many as five guesses may be made in a turn. For each turn, the program will generate a random number between 1 and 36. Each correct guess will be rewarded with points based on how many of your current points you risked.

A game board divides the numbers into rows and columns, as shown in Figure 8-46. This board provides the basis for your guesses.



LOW	01	02	03
	04	05	06
	07	08	09
	10	11	12
MEDIUM	13	14	15
	16	17	18
	19	20	21
	22	23	24
HIGH	25	26	27
	28	29	30
	31	32	33
	34	35	36
	LEFT	**CENTER**	**RIGHT**

FIGURE 8-46 The Guessing Game Board

You can guess whether the random number is even or odd. In this case, you get 1 point for each point risked when you guess correctly. You can guess whether the number is low (1–12), medium (13–24), or high (25–36). In this case, you will get 2 points for each point risked. You can also guess left, center, or right, as shown in Figure 8-46. In this case, you get 2 points for each point risked when your guess is correct. Finally, you can guess a specific number between 0 and 36. In this case you get 36 points for each point risked when your guess is correct.

To make the game more interesting, each round allows up to five guesses. None of the five may be correct, or any number up to all five may be correct. The program stops when the player quits or when the player is out of points.

The program first asks the number of points the user wants to start with, as shown below.

```
How many points would you like? 2000
```

The program then prints the guess menu and allows up to five guesses, as shown in the following example.

```
Guesses Choices
  O   Odd
  E   Even
  L   Low
  M   Med
  H   High
  F   Left
  C   Center
  R   Right
  N   Number

How many guesses would you like?  5
Guess 1
Enter your choice? L
Points at risk?    20

Guess 2    :
Enter your choice? H
Points at risk?    15

Guess 3  :
Enter your choice? N
Enter your number: 18
Points at risk?    20

Guess 4    :
Enter your choice? O
Points at risk?    120

Guess 5    :
Enter your choice? L
Points at risk?    0
```

After all guesses have been made, the program generates the random number and displays the following message:

```
My number is:  31
```

The program then prints the situation of the player:

```
Previous Points: 2000
Guess   Type   Number    Amount    Won/Loss
  1      L                 20         -20
  2      H                 15         +30
  3      N       18        20         -20
  4      O                120        +120
  5      N       20        10         -10
You won 100 points in this turn.
Your new balance is : 2100 points
Do you want to play again (Y or N) ? Y
```

Some special rules: The minimum amount risked on a guess is 0. The maximum is the player's current balance. You need to verify that at no time are the points risked more than the player's current balance. Any combinations of guesses are allowed on a round as long as the total does not exceed the player's balance.

Some hints: You must use at least four arrays, each of five elements. The arrays hold the guess information for the *kind of guess*, *chosen number* (in case the player chooses a number), *amount of the guess*, and *points won or lost*.

Run your program twice, first with 2000 points and then with 500 points. Try different situations. Each run is to exercise each guess at least twice.

48. Write a program to keep records and perform statistical analysis for a class of students. The class may have up to 40 students. There are five quizzes during the term. Each student is identified by a four-digit student number.

The program will print the student scores and calculate and print the statistics for each quiz. The output is in the same order as the input; no sorting is needed. The input will be read from a text file. The output from the program should be similar to the following:

```
  Student     Quiz 1    Quiz 2    Quiz 3    Quiz 4    Quiz 5
   1234         78        83        87        91        86
   2134         67        77        84        82        79
   3124         77        89        93        87        71
 High Score     78        89        93        91        86
 Low Score      67        77        84        82        71
  Average      73.4      83.0      88.2      86.6      78.6
```

Use one- and two-dimensional arrays only. Test your program with the quiz data in Table 8-7.

Student	Quiz 1	Quiz 2	Quiz 3	Quiz 4	Quiz 5
1234	52	7	100	78	34
2134	90	36	90	77	30
3124	100	45	20	90	70
4532	11	17	81	32	77
5678	20	12	45	78	34
6134	34	80	55	78	45
7874	60	100	56	78	78
8026	70	10	66	78	56
9893	34	9	77	78	20
1947	45	40	88	78	55
2877	55	50	99	78	80
3189	22	70	100	78	77
4602	89	50	91	78	60
5405	11	11	0	78	10
6999	0	98	89	78	20

TABLE 8-7 Test Data for Project 48

49. Rework Project 48, creating statistics for each student. Print the students' high, low, and average scores to the right of Quiz 5. Provide appropriate column headings.

50. Program 8-7 builds a frequency array. In the discussion of the algorithm, we noted that a potential problem occurs if any of the data are invalid.

Write a program that uses the random number generator, *rand,* to generate 100 numbers between 1 and 22. The array is then to be passed to a modified version of Program 8-7 that will count the number of values between 0 and 19. Add a 21st element in the array to count all numbers not in the valid range (0 to 19).

Print the input data in a 20 × 5 array—that is, 20 numbers in five rows—and then print the frequency diagram with a heading of the numbers as shown below.

```
--0- --1- --2- --3- --4- … -18- -19- Invalid
```

51. Modify the program you wrote in Project 50 to include a histogram printout of the data. Its format should be similar to Figure 8-15.

52. Using the data from Project 48, build a two-dimensional array of students. Then write a search function that uses the sequential search to find a student in the array and prints the students' scores, average score, and grade based on an absolute scale (90% is A, 80% is B, 70% is C, 60% is D, less than 60% is F). After each printout, give the user the opportunity to continue or stop.

53. Modify the program you wrote in Project 52 to sort the two-dimensional array of students. Then rewrite the search function to use a binary search.

54. Write a program that sorts a 50-element array using the selection sort, the bubble sort, and the insertion sort. Each sort is to be executed twice.

a. For the first sort, fill the array with random numbers between 1 and 1000.

b. For the second sort, fill the array with a nearly ordered list. Create the nearly ordered list by filling the array with sequential numbers and then subtracting 5 from every 10th number in the array.

c. Each sort (selection, bubble, insertion) is to sort the same data. For each sort, count the number of comparisons and moves necessary to order this list.

d. After each sort execution, print the unsorted data followed by the sort data in 5-by-10 matrixes (5 rows of 10 numbers each). After the sorted data, print the number of comparisons and the number of moves required to order the data. Provide appropriate headings for each printout.

e. To make sure your statistics are as accurate as possible, you must analyze each loop limit condition test and each selection statement in your sort functions. The best way to count them is with a comma expression, as shown below. Use similar code for the selection statements.

```
while ((count++, a) && (count++, b))
```

f. Analyze the heuristics you generated, and write a few lines concerning what you discovered about these sorts. Put your comments in a box (asterisks) after your program documentation at the beginning of your program.

55. Write a program to compute the arithmetic mean (average), median, and mode for up to 50 test scores. The data are contained in a text file. The program will also print a histogram of the scores.

The program should start with a function to read the data file and fill the array. Note that there may be fewer than 50 scores. This will require that the read function return the index for the last element in the array.

To determine the average, write a function similar to Program 8-4. To determine the median, you must first sort the array. The median is the score in the middle of the range. This can be determined by selecting the score at `last` / 2 if `last` is an even index and by averaging the scores at the floor and ceiling of `last` / 2 if `last` is odd. The mode is the score that occurs the most often. It can be determined as a byproduct of building the histogram. After building the frequency array, use it to determine which score occurred most often. (Note that two scores can occur the same number of times.)

56. Write a menu-driven program that allows the user to fill an array of 50 integers with random numbers in the range 1...999, sort it, and then search it to determine if a given random number was generated. The menu is shown below.

```
                    MENU
     Select one of the following options:
     F. Fill array with a random number series
     P. Print the array
     S. Sort the array
     Q. Query the array
     Z. Terminate the program
```

- □ Each time the fill option is executed, it will fill the array with a new random number series.
- □ You may use any of the sorts discussed in the text.
- □ If the query locates the number, it will print a message that the number was located and the index location where it was found.
- □ If the query does not locate the number, it will print a message that the number was not in the list and then print the value and index location of the largest number less than the target and the smallest value greater than the requested number. Note that the first and last elements will not have both a smaller and larger value.
- □ If the array has been sorted, the query will use a binary search. If it is not sorted, it will use the sequential search.

To test the program

a. Fill (F) the array.
b. Print (P) the unsorted array.
c. Search (Q) the first, last, and a middle element.
d. Refill (F) the array.
e. Sort (S) the array.
f. Print (P) the sorted array.
g. Search (Q) the first, last, and a middle element.
h. Search (Q) for a number less than the first, greater than the last, and not found in the middle of the list.

Chapter 9
Pointers

Every computer has addressable memory locations. In previous chapters, all of our data manipulations, whether for inspection or alteration, used variable names to locate the data. In other words, we assigned identifiers to data and then manipulated their contents through the identifiers.

Pointers have many uses in C. Besides being a very efficient method of accessing data, they provide efficient techniques for manipulating data in arrays, they are used in functions as pass-by-address parameters, and they are the basis for dynamic allocations of memory.

As shown in Figure 9-1, pointers are the third of the derived types.

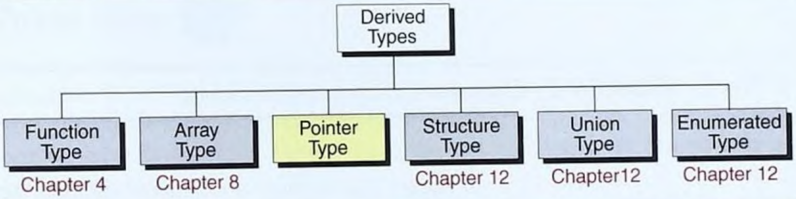


FIGURE 9-1 Derived Types

Objectives

- ❑ To understand the concept and use of pointers
- ❑ To be able to declare, define, and initialize pointers
- ❑ To write programs that access data through pointers
- ❑ To use pointers as parameters and return types
- ❑ To understand pointer compatibility, especially regarding pointers to pointers
- ❑ To understand the role of quality in software engineering

9.1 Introduction

A **pointer** is a constant or variable that contains an address that can be used to access data. Pointers are built on the basic concept of pointer constants. To understand and use pointers, we must first understand this concept.

Pointer Constants

First, let's compare character constants and pointer constants. We know that we can have a character constant, such as any letter of the alphabet, that is drawn from a universe of all characters. In most computers, this universe is ASCII. A character constant is a value and can be stored in a variable. Although the character constant is unnamed, the variable has a name that is declared in the program. This concept is shown in Figure 9-2.

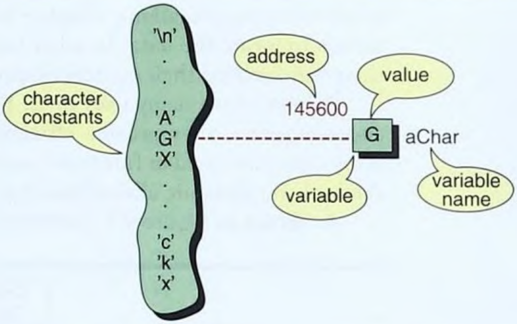


FIGURE 9-2 Character Constants and Variables

In Figure 9-2 we have a character variable, aChar. At this point, aChar contains the value 'G' that was drawn from the universe of character constants. The variable aChar has an address as well as a name. The name is created by the programmer; the address is the relative location of the variable with respect to the program's memory space. Assume, for example, that a computer has only 1 megabyte of memory (2^{20} bytes). Assume also that the computer has chosen the memory location 145600 as the byte to store this variable. This gives us the picture we see in Figure 9-2.

Like character constants, **pointer constants** cannot be changed. In Figure 9-3 we see that the address for our character variable, aChar, was drawn from the set of pointer constants for our computer.

Although the addresses within a computer cannot change, be aware that the address of the variable, aChar, can and will change from one run of our program to another. This is because today's modern operating systems can put a program in memory wherever it is convenient when the program is started. Thus, while aChar is stored at memory location 145600 now, the next time the program is run, it could be located at 876050. It should be obvious, therefore, that even though addresses are constant, we cannot know what they will be. It is still necessary to refer to them symbolically.

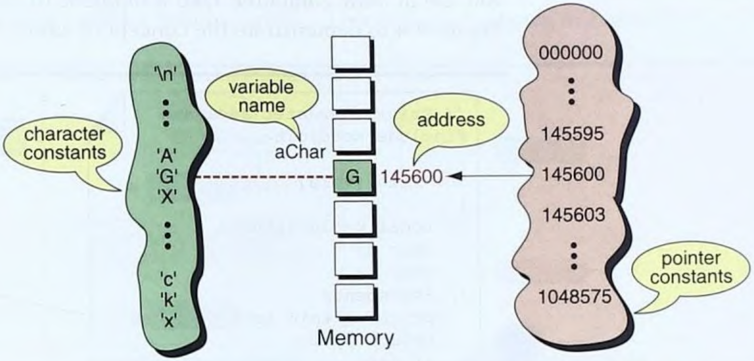


FIGURE 9-3 Pointer Constants

Pointer constants, drawn from the set of addresses for a computer, exist by themselves. We cannot change them; we can only use them.

Pointer Values

Having defined a pointer constant as an address in memory, we now turn our attention to saving this address. If we have a pointer constant, we should be able to save its value if we can somehow identify it. Not only is this possible, but we have been doing it since we wrote our first *scanf* statement with an address operator.

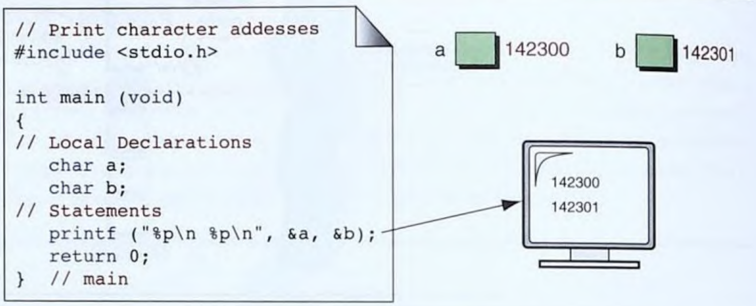
The address operator (&) extracts the address for a variable. The result is a unary expression, which is also known as an address expression. The Precedence Table (see inside front cover) lists the address operator in the unary category, precedence 15.

An address expression, one of the expression types in the unary expression category, consists of an ampersand (&) and a variable name.

The address operator format is seen below.

```
&variable_name
```

Now let's write a program that defines two character variables and prints their addresses as pointers (conversion code %p). Depending on the operating system, this program may print different numbers each time we run it, as previously explained. The addresses would also be different in different computers. However, most of the time, the computer allocates two adjacent memory locations because we defined the two variables one after the other. If you are at your computer, take a moment to code and run the program in Figure 9-4 to demonstrate the concept of address constants.



```
// Print character addesses
#include <stdio.h>

int main (void)
{
// Local Declarations
   char a;
   char b;
// Statements
   printf ("%p\n %p\n", &a, &b);
   return 0;
}  // main
```

FIGURE 9-4 Print Character Addresses

The situation changes slightly when we talk about integers. In most computers, the *int* type occupy 4 bytes. This means that each *int* variable occupies four memory locations. Which of these memory locations is used to find the address of the variable? In most computers, the location of the first byte is used as the memory address. For characters, there is only 1 byte, so its location is the address. For integers, the address is the first byte. This is shown in Figure 9-5.

The same design applies to real and other data types. The address of a variable is the address of the first byte occupied by that variable.

A variable's address is the first byte occupied by the variable.

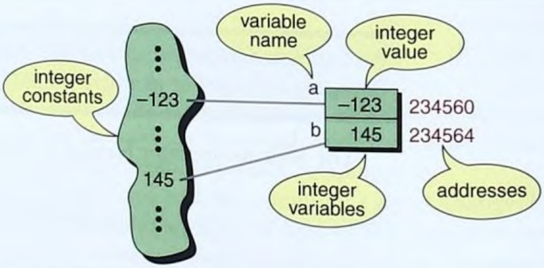


FIGURE 9-5 Integer Constants and Variables

Pointer Variables

If we have a pointer constant and a pointer value, then we can have a **pointer variable**. Thus, we can store the address of a variable into another variable, which is called a pointer variable. This concept is shown in Figure 9-6.

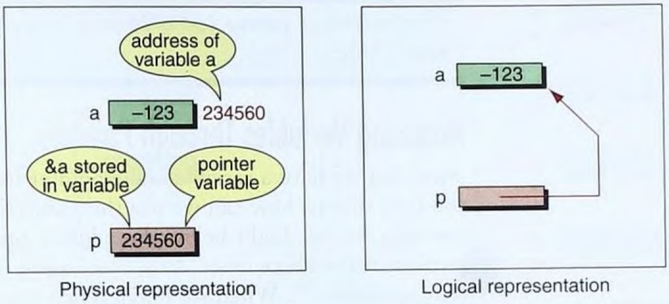


FIGURE 9-6 Pointer Variable

We must distinguish between a variable and its value. Figure 9-6 details the differences. In this figure we see a variable, a, with its value, −123. The variable a is found at location 234560 in memory. Although the variable's name and location are constant, the value may change as the program executes. This figure also has a pointer variable, p. The pointer has a name and a location, both of which are constant. Its value at this point is the memory location 234560. This means that p is pointing to a. In Figure 9-6, the physical representation shows how the data and pointer variables exist in memory. The logical representation shows the relationship between them without the physical details.

We can go even further and store a variable's address in two or more different pointer variables, as is shown in Figure 9-7. This figure has a variable, a, and two pointers, p and q. The pointers each have a name and a location, both of which are constant. Their value at this point is the memory location 234560. This means that both p and q are pointing to a. There is no limit to the number of pointer variables that can point to a variable.

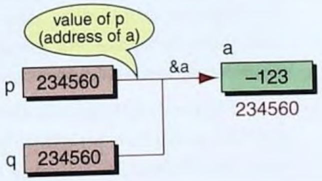


FIGURE 9-7 Multiple Pointers to a Variable

A final thought: If we have a pointer variable, but we don't want it to point anywhere, what is its value? C provides a special null pointer constant, *NULL*, in the standard input/output *stdio.h* header file for this purpose.[1] *NULL* is usually defined as a macro with a value of integer 0 or 0 cast to a void pointer (`void*`).

A pointer that points to no variable contains the special null-pointer constant, *NULL*.

Accessing Variables Through Pointers

Now that we have a variable and a pointer to the variable, how can we relate the two; that is, how can we use the pointer? Once again C has provided an operator for us. Right below the address operator in the unary expressions portion of the Precedence Table (see inside front cover), we find the indirection operator (*). When we dereference a pointer, we are using its value to reference (address) another variable. The indirection operator is a unary operator whose operand must be a pointer value. The result is an expression that can be used to access the pointed variable for the purpose of inspection or alteration. To access the variable `a` through the pointer `p`, we simply code `*p`. The indirection operator is shown below.

```
*p
```

Let's assume that we need to add 1 to the variable `a`. We can do this with any of the following statements, assuming that the pointer, `p`, is properly initialized (`p = &a`).

```
a++;    a = a + 1;    *p = *p + 1;    (*p)++;
```

In the last example, `(*p)++`, we need the parentheses. The postfix increment has a precedence of 16 in the Precedence Table while indirection, which is a unary operator, has a precedence of 15. The parentheses therefore force the dereference to occur before the addition so that we add to the data variable and not to the pointer. Without the parentheses, we would add to the pointer first, which would change the address.

An indirect expression, one of the expression types in the unary expression category, is coded with an asterisk (`*`) and an identifier.

Figure 9-8 expands the discussion. Let's assume that the variable `x` is pointed to by two pointers, `p` and `q`. The expressions `x`, `*p`, `*q` all are expressions that allow the variable to be either inspected or changed. When they are

[1] Technically, `NULL` is found in the *stddef.h* library. Traditionally, it may also be defined in *stdio.h*.

used in the right-hand side of the assignment operator, they can only inspect (copy or compare). When they are used in the left-hand side of the assignment operator, they change the value of x.

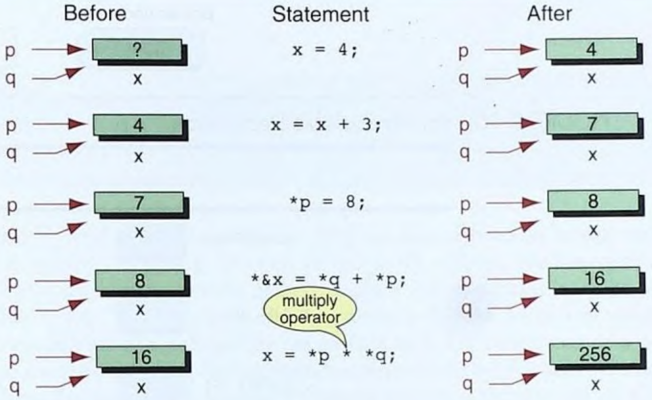


FIGURE 9-8 Accessing Variables Through Pointers

The indirection and address operators are the inverse of each other, and when combined in an expression, such as `*&x`, they cancel each other. To see this, let's break down the expression. These two unary operators are evaluated from the right. The first expression is therefore `&x`, the address of x, which as we have seen, is a pointer value. The second expression, `*(&x)`, dereferences the pointer constant, giving the variable (x) itself. Therefore, the operators effectively cancel each other (Figure 9-9). Of course, we would never code the expression `*&a` in a program; we use it in Figure 9-8 for illustration only.

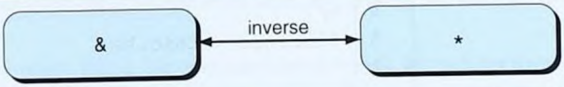


FIGURE 9-9 Address and Indirection Operators

Pointer Declaration and Definition

As shown in Figure 9-10, we use the asterisk to declare pointer variables. When we use the asterisk in this way, it is really not an operator but rather a compiler syntactical notation.

Figure 9-11 shows how we declare different pointer variables. Their corresponding data variables are shown for comparison. Note that in each case the pointer is declared to be of a given type. Thus, p is a pointer to characters, while q is a pointer to integers, and r is a pointer to floating-point variables.

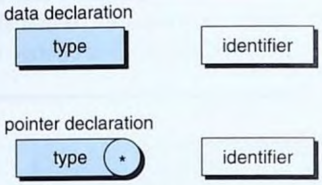


FIGURE 9-10 Pointer Variable Declaration

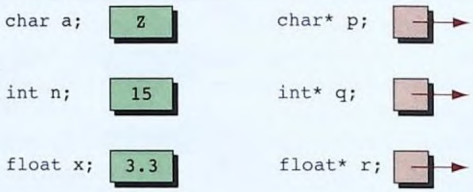


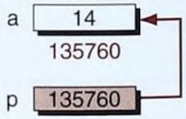
FIGURE 9-11 Declaring Pointer Variables

EXAMPLE 9-1 Print with a Pointer

Program 9-1 stores the address of a variable in a pointer and then prints the data using the variable value and a pointer.

PROGRAM 9-1 Demonstrate Use of Pointers

```
/* Demonstrate pointer use
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int  a;
   int* p;

// Statements
   a = 14;
   p = &a;

   printf("%d %p\n", a, &a);
```



continued

PROGRAM 9-1 Demonstrate Use of Pointers *(continued)*

```
   printf("%p %d %d\n",
          p, *p, a);

   return 0;
}  // main
```

```
Results:
14 00135760
00135760 14 14
```

Program 9-1 Analysis Program 9-1 requires a little explanation. First, we have defined an integer variable, a, and a pointer to integer, p, to which we assign a's address. We then print twice. The first print displays the contents of the variable a and its address (note the pointer conversion code). The second print uses the pointer, p. It prints the pointer value containing the address of a, followed by the contents of a, first referenced as a pointer and then as a variable. This demonstrates two ways to access the data. We suggest that you run this program for yourself. Of course, when you do, you will get a different address. Also, some compilers display addresses as integers, and some use hexadecimal numbers.

Declaration versus Redirection

We have used the asterisk operator in two different contexts: for declaration and for redirection. When as asterisk is used for declaration, it is associated with a type. For example, we define a pointer to an integer as

```
int* pa;
int* pb;
```

In this usage, the asterisk declares that the type of pa and pb is a pointer to integer.

On the other hand, we also use the asterisk for redirection. When used for redirection, the asterisk is an operator that redirects the operation from the pointer variable to a data variable. For example, given two pointers to integers, pa and pb, sum is computed as

```
sum = *pa + *pb;
```

The first expression, *pa, uses the pointer value stored in the pointer variable, pa, to locate its value. The address in pa is not added; it is *redirected* through the asterisk operator to the desired value. Likewise, the address in the second expression, pb, is not added but used to locate the second expression's value. Once both of the expressions have been evaluated, they can be added and the resulting expression value stored in sum.

Initialization of Pointer Variables

As we discussed in the previous section, the C language does not initialize variables. Thus, when we start our program, all of our uninitialized variables have unknown garbage in them.

The same is true for pointers. When the program starts, uninitialized pointers will have some unknown memory address in them. More precisely, they will have an unknown value that will be interpreted as a memory location. Most likely the value will not be valid for the computer you are using, or if it is, will not be valid for the memory you have been allocated. If the address does not exist, you will get an immediate run-time error. If it is a valid address, you often, but unfortunately not always, get a run-time error. (It is better to get the error when you use the invalid pointer than to have the program produce garbage.)

One of the most common causes of errors in programming, by novices and professionals alike, is uninitialized pointers. These errors can be very difficult to debug because the effect of the error is often delayed until later in the program execution. Figure 9-12 shows both an uninitialized variable and an uninitialized pointer.

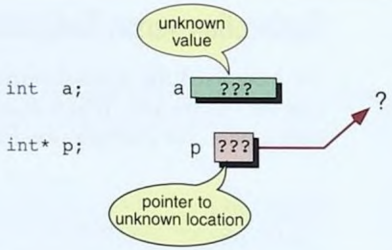


FIGURE 9-12 Uninitialized Pointers

To correct the problem shown in Figure 9-12, we must always assign a valid memory address to the pointer, as shown in the next example.

```
int  a;                      // int variable-value unknown
int* p = &a;                 // p has valid address
*p = 89;                     // a assigned value 89
```

As we saw with variables, it is also possible to initialize pointers when they are declared and defined. All that is needed is that the data variable be defined before the pointer variable. For example, if we have an integer variable, a, and a pointer to integer, p, then to set p to point to a at declaration time, we can code it as shown in Figure 9-13.

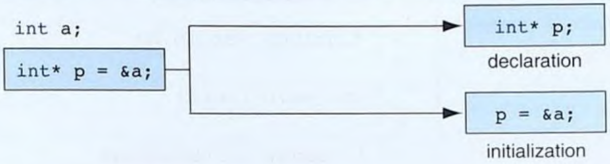


FIGURE 9-13 Initializing Pointer Variables

Note that in Figure 9-13, the initialization involves two different steps. First the variable is declared. Then the assignment statement to initialize the variable is generated. Some style experts suggest that you should not use an initializer in this way.[2] Their argument is that it saves no code; that is, that the initializer statement is required either as a part of the declaration and initialization or as a separately coded statement in the statement section of the function. Putting the initialization in the declaration section tends to hide it and make program maintenance more difficult. We tend to agree with them.

We can also set a pointer to NULL, either during definition or during execution. The following statement demonstrates how we could define a pointer with an initial value of NULL.

```
int* p = NULL;
```

In most implementations, a null pointer contains address 0, which may be a valid or invalid address depending on the operating system. If we dereference the pointer p when it is NULL, we will most likely get a run-time error because NULL is not a valid address. The type of error we get depends on the system we are using.

EXAMPLE 9-2 Change Variables—Fun with Pointers

Now let's write a program and have some fun with pointers. Our code is shown in Program 9-2. Do not try to figure out why this program is doing what it is doing; there is no reason. Rather, just try to trace the different variables and pointers as we change them.

PROGRAM 9-2 Fun with Pointers

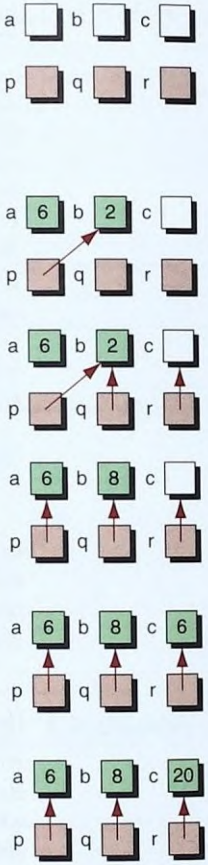
```
/* Fun with pointers
      Written by:
      Date:
*/
```

continued

[2] For example, see Steve Oualline, *C Elements of Style* (Mountain View, Calif.: M&T Books, 1992).

PROGRAM 9-2 Fun with Pointers *(continued)*

```
#include <stdio.h>

int main (void)
{
// Local Declarations
   int  a;
   int  b;
   int  c;
   int* p;
   int* q;
   int* r;

// Statements
   a = 6;
   b = 2;
   p = &b;

   q = p;
   r = &c;

   p  = &a;
   *q = 8;

   *r = *p;

   *r = a + *q + *&c;

   printf("%d %d %d \n",
           a, b, c);
   printf("%d %d %d",
         *p, *q, *r);
   return 0;
} // main
```



```
Results:
6 8 20
6 8 20
```

Program 9-2 Analysis When the program starts, all the variables and their pointers are uninitialized. The variables have garbage values, and the pointers have invalid memory locations.

The first thing the program does, therefore, is to assign values to a and b, and to initialize all three pointers. After statement 23, both p and q point to b, and r points to c.

Statement 25 assigns the address of a to p. All three pointers now point to different variables. Using the indirection operator, we dereference q (*q), and assign b the value 8.

Statement 28 demonstrates that both operands can be dereferenced when it assigns the contents of a (*p) to c (*r).

Finally, in statement 30, we use three different formats to sum the values in the variables, a variable name, a dereferenced pointer, and a dereferenced address operator. Using the figures in the program, trace these statements carefully to assure yourself that you understand how they work.

EXAMPLE 9-3 Add Two Numbers

This example shows how we can use pointers to add two numbers. It explores the concept of using pointers to manipulate data. A graphic representation of the variables is shown in Figure 9-14. The code is shown in Program 9-3.

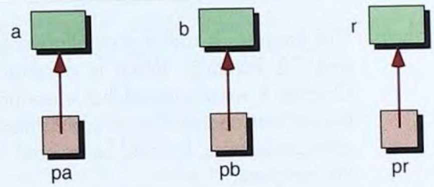


FIGURE 9-14 Add Two Numbers Using Pointers

PROGRAM 9-3 Add Two Numbers Using Pointers

```
/* This program adds two numbers using pointers to
   demonstrate the concept of pointers.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int  a;
   int  b;
   int  r;
   int* pa = &a;
```

continued

PROGRAM 9-3 Add Two Numbers Using Pointers *(continued)*

```
15      int* pb = &b;
16      int* pr = &r;
17
18   // Statements
19      printf("Enter the first number : ");
20      scanf ("%d", pa);
21      printf("Enter the second number: ");
22      scanf ("%d", pb);
23      *pr = *pa + *pb;
24      printf("\n%d + %d is %d", *pa, *pb, *pr);
25      return 0;
26   } // main
```

```
Results:
   Enter the first number : 15
   Enter the second number: 51

   15 + 51 is 66
```

Program 9-3 Analysis This program is rather straightforward except for one thing. Look at statements 20 and 22 carefully. What is missing? When we discussed pass-by address in Chapter 4, we mentioned that when input areas are passed by address, we don't use the address operator in the *scanf* statement. This is because the pointer already contains an address. It would be an error to use the address operator on the pointers in this program.

EXAMPLE 9-4 Pointer Flexibility

This example shows how we can use the same pointer to print the value of different variables. The variables and their pointer are shown in Figure 9-15.

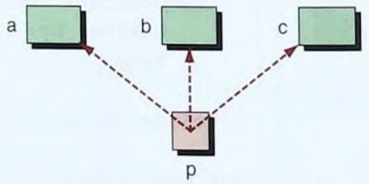


FIGURE 9-15 Demonstrate Pointer Flexibility

The code is in Program 9-4.

PROGRAM 9-4 Using One Pointer for Many Variables

```
/* This program shows how the same pointer can point to
   different data variables in different statements.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int  a;
   int  b;
   int  c;
   int* p;

// Statements
   printf("Enter three numbers and key return: ");
   scanf ("%d %d %d", &a, &b, &c);
   p = &a;
   printf("%3d\n", *p);
   p = &b;
   printf("%3d\n", *p);
   p = &c;
   printf("%3d\n", *p);
   return 0;
}  // main
```

```
Results:
Enter three numbers and key return: 10 20 30
 10
 20
 30
```

EXAMPLE 9-5 Multiple Pointers For One Variable

This example shows how we can use different pointers to print the value of the same variable. The variable and its pointers are shown in Figure 9-16.

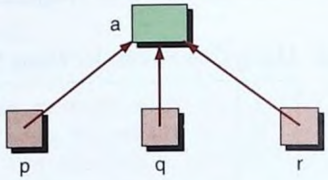


FIGURE 9-16 One Variable with Many Pointers

The code is shown in Program 9-5.

PROGRAM 9-5 Using A Variable with Many Pointers

```
/* This program shows how we can use different pointers
   to point to the same data variable.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int  a;
   int* p = &a;
   int* q = &a;
   int* r = &a;

// Statements
   printf("Enter a number: ");
   scanf ("%d", &a);
   printf("%d\n", *p);
   printf("%d\n", *q);
   printf("%d\n", *r);

   return 0;
}  // main
```

```
Results:
Enter a number: 15
15
15
15
```

9.2 Pointers for Inter-function Communication

One of the most useful applications of pointers is in functions. When we discussed functions in Chapter 4, we saw that C uses the pass-by-value for downward communication. For upward communication, the only direct way to send something back from a function is through the *return* value. We also saw that we can use upward and bi-directional communication by passing an address and using it to refer back to data in the calling program. When we pass an address, we are actually passing a pointer to a variable. In this section, we fully develop the bi-directional communication. We use two examples to demonstrate how pointers can be used in functions.

Passing Addresses

Figures 9-17 and 9-18 demonstrate the use of pointers. In Figure 9-17, we call the exchange function, passing it two variables whose values are to be exchanged. When we use downward communication, the data are exchanged in the called function, but nothing changes in the calling program.

Rather than passing data to the called function, we need to pass addresses (pointers). Once we have a pointer to a variable, it doesn't make any difference if it is local to the active function, defined in *main*, or even if it is a global variable—we can change it! In other words, given a pointer to a variable anywhere in our memory space, we can change the contents of the variable.

Figure 9-18 shows the exchange using pointers. To create the pointers, we use the address operator (`&`) in the call, as shown below. We are now passing the address of `a` and `b` rather than their values.

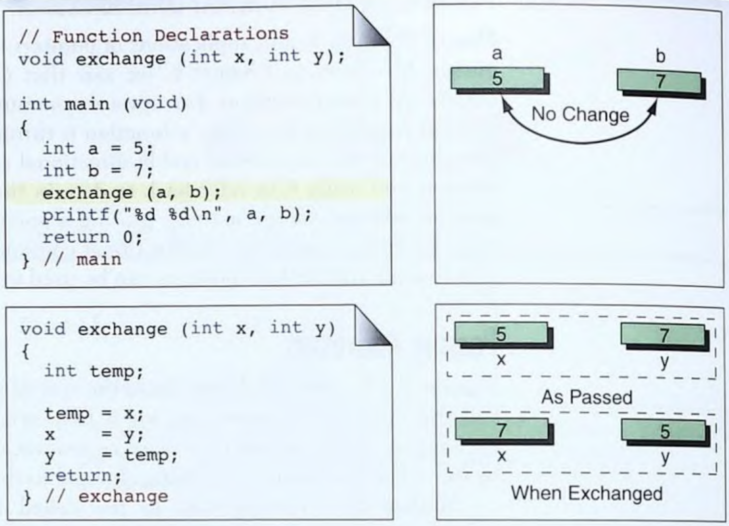
```
exchange (&a, &b);
```

To pass addresses, the formal parameters in the called function are defined as a pointer to variables. This definition, which is shown below, completes the connection and allows us to indirectly reference the variables in the calling program through the pointers.

```
void exchange (int* px, int* py);
```

To assign a value to `a`, all that we need to use is the indirection operator in `exchange`. Using the indirection operator allows us to access data in the calling program.

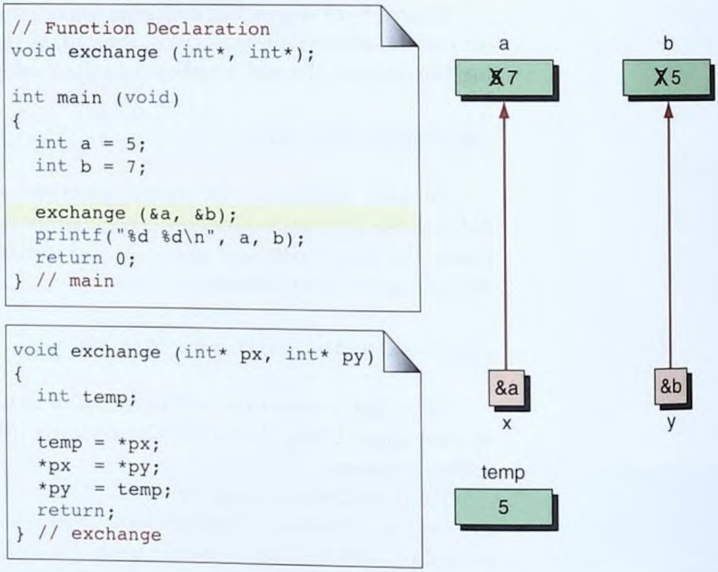
We then call `exchange` using the address operator for the variables that we want to exchange. Note that `exchange` uses the two formal parameters, `px` and `py`, and one local variable, `temp`. By dereferencing the parameters, we make the exchange using the variables in *main* and the local variable, `temp`, in `exchange`. The correct logic is shown in Figure 9-18.



```
// Function Declarations
void exchange (int x, int y);

int main (void)
{
  int a = 5;
  int b = 7;
  exchange (a, b);
  printf("%d %d\n", a, b);
  return 0;
} // main
```

```
void exchange (int x, int y)
{
  int temp;

  temp = x;
  x    = y;
  y    = temp;
  return;
} // exchange
```

FIGURE 9-17 An Unworkable Exchange



```
// Function Declaration
void exchange (int*, int*);

int main (void)
{
  int a = 5;
  int b = 7;

  exchange (&a, &b);
  printf("%d %d\n", a, b);
  return 0;
} // main
```

```
void exchange (int* px, int* py)
{
  int temp;

  temp = *px;
  *px  = *py;
  *py  = temp;
  return;
} // exchange
```

FIGURE 9-18 Exchange Using Pointers

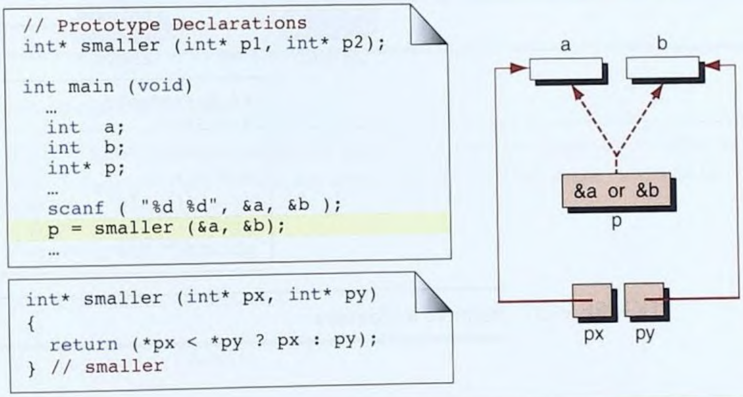
In summary, when we need to send more than one value back from a function, we use pointers. Using either upward or bi-directional communication, we can change any number of variables.

> **Every time we want a called function to have access to a variable in the calling function, we pass the address of that variable to the called function and use the indirection operator to access it.**

Functions Returning Pointers

We have shown you many examples of functions using pointers, but so far we have shown none that return a pointer. Nothing prevents a function from returning a pointer to the calling function. In fact, as you will see, it is quite common for functions to return pointers.

As an example, let's write a rather trivial function to determine the smaller of two numbers. In this case, we need a pointer to the smaller of two variables, a and b. Since we are looking for a pointer, we pass two pointers to the function, which uses a conditional expression to determine which value is smaller. Once we know the smaller value, we can return the address of its location as a pointer. The return value is then placed in the calling function's pointer, p, so that after the call it points to either a or b, based on their values. Both the code and a diagram of the variables and pointers are seen in Figure 9-19.



```
// Prototype Declarations
int* smaller (int* p1, int* p2);

int main (void)
  …
  int  a;
  int  b;
  int* p;
  …
  scanf ( "%d %d", &a, &b );
  p = smaller (&a, &b);
  …
```

```
int* smaller (int* px, int* py)
{
  return (*px < *py ? px : py);
} // smaller
```

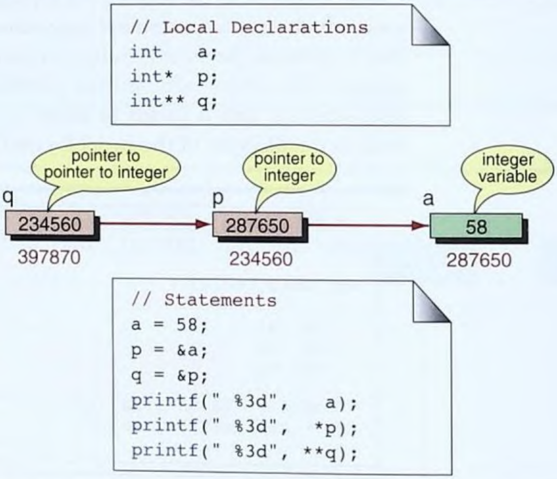
FIGURE 9-19 Functions Returning Pointers

When we return a pointer, it must point to data in the calling function or a higher level function. It is an error to return a pointer to a local variable in the called function, because when the function terminates its memory may be used by other parts of the program. Although a simple program might not notice the error because the space was not reused, a large program would either get the wrong answer or fail when the memory being referenced by the pointer was changed.

It is a serious error to return a pointer to a local variable.

9.3 Pointers to Pointers

So far, all our pointers have been pointing directly to data. It is possible—and with advanced data structures often necessary—to use pointers that point to other pointers. For example, we can have a pointer pointing to a pointer to an integer. This two-level indirection is seen in Figure 9-20. There is no limit as to how many levels of indirection we can use but, practically, we seldom use more than two.



```
// Local Declarations
int   a;
int*  p;
int** q;
```

```
// Statements
a = 58;
p = &a;
q = &p;
printf(" %3d",   a);
printf(" %3d",  *p);
printf(" %3d", **q);
```

FIGURE 9-20 Pointers to Pointers

Each level of **pointer indirection** requires a separate indirection operator when it is dereferenced. In Figure 9-20, to refer to a using the pointer p, we have to dereference it once, as shown below.

```
*p
```

To refer to the variable a using the pointer q, we have to dereference it twice to get to the integer a because there are two levels of indirection (pointers) involved. If we dereference it only once, we are referencing p, which is a pointer to an integer. Another way to say this is that q is a pointer to a pointer to an integer. The double dereference is shown below.

```
**q
```

Figure 9-20 also shows how we use these concepts in a C code fragment. All three references in the *printf* statements refer to the variable a. The first print statement prints the value of the variable a directly; the second uses the pointer p; the third uses the pointer q. The result is the value 58 printed three times, as shown below.

```
58   58   58
```

EXAMPLE 9-6 Pointer-to-Pointer

The last example in this section shows how we can use different pointers with pointers to pointers and pointers to pointers to pointers to read the value of the same variable. A graphic representation of the variables is shown in Figure 9-21. The code is shown in Program 9-6.

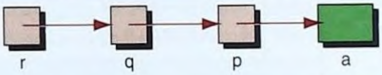


FIGURE 9-21 Using Pointers to Pointers

PROGRAM 9-6 Using pointers to pointers

```
/* Show how pointers to pointers can be used by different
   scanf functions to read data to the same variable.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int    a;
   int*   p;
   int**  q;
   int*** r;

// Statements
   p = &a;
```

continued

PROGRAM 9-6 Using pointers to pointers *(continued)*

```
   q = &p;
   r = &q;

   printf("Enter a number: ");
   scanf ("%d", &a);                              // Using a
   printf("The number is : %d\n", a);

   printf("\nEnter a number: ");
   scanf ("%d", p);                               // Using p
   printf("The number is : %d\n", a);

   printf("\nEnter a number: ");
   scanf ("%d", *q);                              // Using q
   printf("The number is : %d\n", a);

   printf("\nEnter a number: ");
   scanf ("%d", **r);                             // Using r
   printf("The number is : %d\n", a);

   return 0;
} // main
```

```
Results:
Enter a number: 1
The number is : 1

Enter a number: 2
The number is : 2

Enter a number: 3
The number is : 3

Enter a number: 4
The number is : 4
```

Program 9-6 Analysis In each successive read, we use a higher level of indirection. In the print statements, however, we always use the integer variable, a, to prove that the reads were successful.

9.4 Compatibility

It is important to recognize that pointers have a type associated with them. They are not just pointer types, but rather are pointers to a *specific* type, such as character. Each pointer therefore takes on the attributes of the type to which it refers in addition to its own attributes.

Pointer Size Compatibility

The size of all pointers is the same. Every pointer variable holds the address of one memory location in the computer. On the other hand, the size of the variable that the pointer references can be different; it takes the attributes of the type being referenced. This is demonstrated by Program 9-7, which prints the size of each pointer and the size of the referenced object.

PROGRAM 9-7 Demonstrate Size of Pointers

```
/* Demonstrate size of pointers.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   char  c;
   char* pc;
   int   sizeofc      = sizeof(c);
   int   sizeofpc     = sizeof(pc);
   int   sizeofStarpc = sizeof(*pc);

   int  a;
   int* pa;
   int  sizeofa      = sizeof(a);
   int  sizeofpa     = sizeof(pa);
   int  sizeofStarpa = sizeof(*pa);

   double  x;
   double* px;
   int sizeofx      = sizeof(x);
   int sizeofpx     = sizeof(px);
   int sizeofStarpx = sizeof(*px);

// Statements
   printf("sizeof(c): %3d | ",  sizeofc);
   printf("sizeof(pc): %3d | ", sizeofpc);
   printf("sizeof(*pc): %3d\n", sizeofStarpc);

   printf("sizeof(a): %3d | ",  sizeofa);
   printf("sizeof(pa): %3d | ", sizeofpa);
   printf("sizeof(*pa): %3d\n", sizeofStarpa);

```

continued

PROGRAM 9-7 Demonstrate Size of Pointers *(continued)*

```
   printf("sizeof(x): %3d | ",  sizeofx);
   printf("sizeof(px): %3d | ", sizeofpx);
   printf("sizeof(*px): %3d\n", sizeofStarpx);

   return 0;
}  // main
```

```
Results:
sizeof(c):   1 | sizeof(pc):   4 | sizeof(*pc):   1
sizeof(a):   4 | sizeof(pa):   4 | sizeof(*pa):   4
sizeof(x):   8 | sizeof(px):   4 | sizeof(*px):   8
```

Program 9-7 Analysis

What is this code telling us? First, note that the variables a, c, and x are never assigned values. This means that the sizes are independent of whatever value may be in a variable. In other words, the sizes are dependent on the type and not its values.

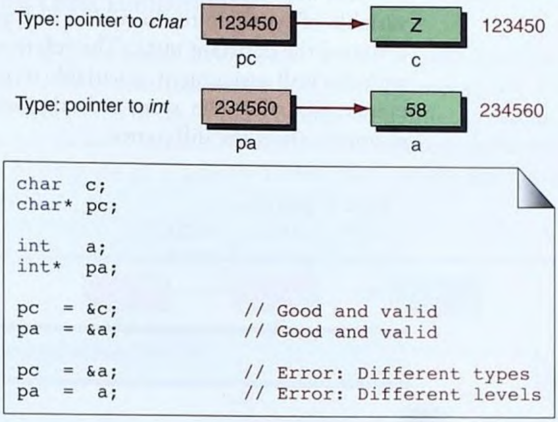
Now look at the size of the pointers. It is 4 in all cases, which is the size of an address in the computer on which this program was run. This makes sense: All computers today have more than 65,535 bytes, which is the maximum address that could be stored in 2 bytes.

But note what happens when we print the size of the type that the pointer is referring to: The size is the same as the data size! This means that in addition to the size of the pointer, the system also knows the size of whatever the pointer is pointing to. To confirm this, look at the size of the pointer, px, and what it is pointing to when dereferenced (*px).

Dereference Type Compatibility

The second issue in compatibility is the dereference type. The dereference type is the type of the variable that the pointer is referencing. With one exception (discussed later), it is invalid to assign a pointer of one type to a pointer of another type, even though the values in both cases are memory addresses and would therefore seem to be fully compatible. Although the addresses may be compatible because they are drawn from the same set, what is not compatible is the underlying data type of the referenced object.

In C, we can't use the assignment operator with pointers to different types; if we try to, we get a compile error. A pointer to a *char* is only compatible with a pointer to a *char*; and a pointer to an *int* is only compatible with a pointer to an *int*. We cannot assign a pointer to a *char* to a pointer to an *int*. Let's construct an example in which we have two variables: one *int* and one *char*. We also define one pointer to *char* and one pointer to *int* as shown in Figure 9-22.



```
char  c;
char* pc;

int   a;
int*  pa;

pc   = &c;              // Good and valid
pa   = &a;              // Good and valid

pc   = &a;              // Error: Different types
pa   =  a;              // Error: Different levels
```

FIGURE 9-22 Dereference Type Compatibility

The first pair of assignments are valid. In the first assignment, we store the address of a character variable in a pointer to character variable. In the second assignment, we store the address of an integer (*int*) variable in a pointer to an integer (*int*) variable. We create an error in the third assignment because we try to store the address of a character variable into a pointer variable whose type is pointer to integer (*int*). We also get an error in the fourth assignment.

Pointer to Void

The exception to the reference type compatibility rule is the *pointer to void*. As we discussed in Chapter 2, one of the types defined in C is *void*. Although the *void* type defines the lack of any other type when used with functions, its use with pointers is different. A pointer to *void* is a generic type that is not associated with a reference type; that is, it is not the address of a character, an integer, a real, or any other type. However, it is compatible *for assignment purposes only* with all other pointer types. Thus, a pointer of any reference type can be assigned to a pointer to *void* type and a pointer to *void* type can be assigned to a pointer of any reference type. There is, however, one restriction; since a *void* pointer has no object type, it cannot be dereferenced unless it is cast (see next section). The following shows how we can declare a variable of pointer to *void* type.

```
void* pVoid;
```

It is important to understand the difference between a null pointer and a variable of type pointer to *void*. A null pointer is a pointer of any type that is assigned the constant NULL. The reference type of the pointer will not change with the null assignment. A variable of pointer to *void* is a pointer with no reference type that can store only the address of any variable. The following examples show the difference

```
void* pVoid;                    // Pointer to void type
int*  pInt  = NULL;             // Null pointer of type int
char* pChar = NULL;             // Null pointer of type char
```

A void pointer cannot be dereferenced.

Casting Pointers

The problem of type incompatibility can be solved if we use casting. We can make an explicit assignment between incompatible pointer types by using a cast, just as we can cast an *int* to a *char*. For example, if we decided that we needed to use the *char* pointer, pc in the previous example, to point to an *int* (a), we could cast it as shown below.

```
pc = (char*)&a;
```

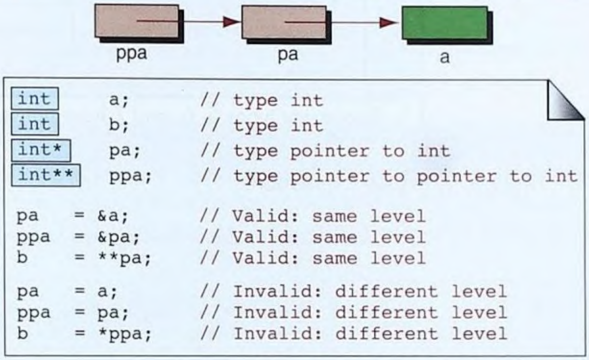
But in this case, *user beware!* Unless we cast all operations that use pc, we have a great chance of creating serious errors. In fact, with the exception of the *void* pointer, we should never cast a pointer. The following assignments are all valid, but they are extremely dangerous and must be used with a very carefully thought-out design.

```
// Local Declarations
   void* pVoid;
   char* pChar;
   int*  pInt;
// Statements
   pVoid  = pChar;
   pInt   = pVoid;
   pInt   = (int*) pChar;
```

Another use of the cast is to provide a type for the *void* pointer. As noted in the previous section, a *void* pointer cannot be dereferenced because it has no object type. When we cast it, however, we provide the type.

Dereference Level Compatibility

Compatibility also includes dereference level compatibility. For example, a pointer to *int* is not compatible with a pointer-to-pointer to *int*. The pointer to *int* has a reference type of *int*, while a pointer-to-pointer to *int* has a reference type of pointer to *int*. Figure 9-23 shows two pointers declared at different levels. The pointer pa is a pointer to *int*; the pointer ppa is a pointer-to-pointer to *int*.



```
int      a;      // type int
int      b;      // type int
int*     pa;     // type pointer to int
int**    ppa;    // type pointer to pointer to int

pa   = &a;       // Valid: same level
ppa  = &pa;      // Valid: same level
b    = **pa;     // Valid: same level

pa   = a;        // Invalid: different level
ppa  = pa;       // Invalid: different level
b    = *ppa;     // Invalid: different level
```

FIGURE 9-23 Dereference Level Compatibility

9.5 Lvalue and Rvalue

In C, an expression is either an lvalue or an rvalue. As you know, every expression has a value. But the value in an expression (after evaluation) can be used in two different ways.

1. An **lvalue** expression must be used whenever the object is receiving a value; that is, it is being modified.

2. An **rvalue** expression can be used to supply a value for further use; that is, to examine or copy its value.

But how do you know when an expression is an lvalue and when it is an rvalue? Fortunately, only seven types of expressions are lvalue expressions. They are shown in Table 9-1.

Expression Type[a]		Comments
1.	identifier	Variable identifier
2.	expression[...]	Array indexing
3.	(expression)	Expression must already be lvalue
4.	*expression	Dereferenced expression
5.	expression.name	Structure selection
6.	expression->name	Structure indirect selection
7.	function call	If function uses *return* by address

[a] Expression types 5, 6, and 7 have not yet been covered.

TABLE 9-1 lvalue Expressions

For example, the following are lvalue expressions:

```
a = ...      a[5] = ...      (a) = ...      *p = ...
```

All expressions that are not lvalue expressions are rvalues. The following show some rvalue expressions:

```
5      a + 2      a * 6      a[2] + 3      a++
```

Note that even if an expression is an lvalue, if it is used as part of a larger expression in which the operators create only rvalue expressions, then the whole expression is an rvalue. For example, a[2] is an lvalue. But when it is used in the expression a[2] + 3, the whole expression is an rvalue, not an lvalue. Similarly, in the expression a++, the variable a is an lvalue while the whole expression (a++) is an rvalue.

You may ask, "Why worry so much about lvalues and rvalues?" The reason is that some operators need an lvalue as their operand. If we use one of these operators and use an rvalue in place of the operand, we will get a compile error.

The right operand of an assignment operator must be an rvalue expression.

Only six operators need an lvalue expression as an operand: address operator, postfix increment, postfix decrement, prefix increment, prefix decrement, and assignment. They are shown in Table 9-2 with examples of each.

Type of Expression	Examples	
Address operator	`&score`	
Postfix increment/decrement	`x++`	`y--`
Prefix increment/decrement	`++x`	`--y`
Assignment (left operand)	`x = 1`	`y += 3`

TABLE 9-2 Operators That Require lvalue Expressions

Table 9-3 contains several examples of invalid expressions that will create syntax errors because an rvalue is used when an lvalue is needed.

Expression	Problem
`a + 2 = 6;`	a + 2 is an rvalue and cannot be the left operand in an assignment; it is a temporary value that does not have an address; no place to store 6.
`&(a + 2);`	a + 2 is an rvalue, and the address operator needs an lvalue; rvalues are temporary values and do not have addresses.
`&4;`	Same as above (4 is an rvalue).

`(a + 2)++;`
`++(a + 2);`
	Postfix and prefix operators require lvalues; (a + 2) is an rvalue.

TABLE 9-3 Invalid rvalue Expressions

One final thought: A variable name can assume the role of either an lvalue or an rvalue depending on how it is used in an expression. In the following expression, a is an lvalue because it is on the left of the assignment and b is an rvalue because it is on the right of the assignment.

```
a = b
```

Pointer Examples

In this section we demonstrate two ways we can use pointers when calling functions.

EXAMPLE 9-7 Convert Seconds to Hours

We begin with a simple function that converts time in seconds to hours, minutes, and seconds. While the function is simple, it does require three address parameters to return the values. The code is shown in Program 9-8.

PROGRAM 9-8 Convert Seconds to Hours, Minutes, and Seconds

```
/* ================== secToHours ==================
   Given time in seconds, convert it to hours, minutes,
   and seconds.
      Pre    time in seconds
             addresses of hours, minutes, seconds
      Post   hours, minutes, seconds calculated
      Return error indicator--1 success, 0 bad time
*/
int secToHours (long time,
                int* hours, int* minutes, int* seconds)
{
// Local Declarations
   long localTime;

// Statements
   localTime = time;
   *seconds  = localTime % 60;
   localTime = localTime / 60;

   *minutes  = localTime % 60;

   *hours    = localTime / 60;

   if (*hours > 24)
      return 0;
   else
      return 1;
}  // secToHours
```

Program 9-8 Analysis The first question you might ask when reading this simple function is "Why define a local variable for the time?" In this short function, it really wasn't necessary. However, a good programmer does not change a value parameter within the function, because its original value may be needed later. We have seen times when "later" turned out to be a maintenance change and hours were spent debugging the error when the wrong value was computed.

Create local variables when a value parameter will be changed within a function so that the original value will always be available for processing.

Another important design point: If you need to send back two values from a called function, *do not pass one back through a pointer and return the other.* Either use the return for some other reason, such as a status flag, or make the return *void*. Keeping your design consistent—in this case by using a consistent method of returning the values—makes your programs easier to understand and is an example of the KISS principle.

When several values need to be sent back to the calling function, use address parameters for all of them. Do not return one value and use address parameters for the others.

EXAMPLE 9-8 Quadratic Equations

Let's look at a typical program design that reads, processes, and prints data. This is a processing cycle that is common to many, many programs. Figure 9-24 shows its structure chart.

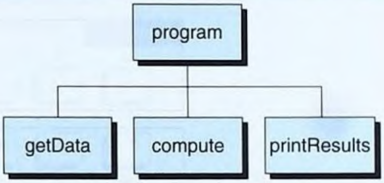


FIGURE 9-24 A Common Program Design

To demonstrate the universality of this design, let's compute the real roots for a quadratic equation. Recall that a quadratic equation has the form

$$\mathbf{ax^2 + bx + c = 0}$$

Four possible situations can occur when you solve for the roots in a quadratic equation. First, it is an error if both a and b are 0: There is no solution. Second, if a is 0 and b is not 0, there is only one root:

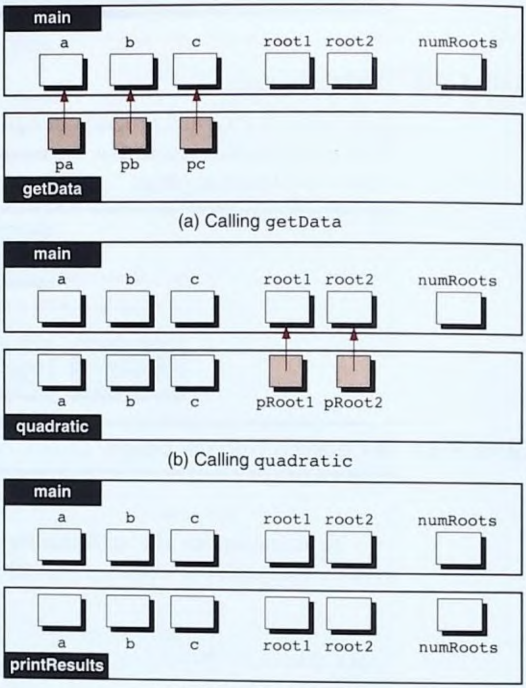
$$\mathbf{x = \frac{-c}{b}}$$

Third, if $b^2 - 4ac$ is 0 or positive, there are two, possibly equal, roots derived from the following equation:

$$\mathbf{x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}}$$

Finally, if $b^2 - 4ac$ is negative, the roots are imaginary.

Figure 9-25 diagrams the interaction of the variables and pointers for Program 9-9. In this short program, we use pointers to pass data from a read function, pass both values and pointers to a compute function, and finally pass the values to a print function.



(a) Calling `getData`

(b) Calling `quadratic`

(c) Calling `printResults`

FIGURE 9-25 Using Pointers as Parameters

PROGRAM 9-9 Quadratic Roots

```
/* Test driver for quadratic function.
      Written by:
      Date:
*/
#include <stdio.h>
#include <math.h>

// Function Declarations
void getData      (int*    pa,      int* pb, int* pc);
int  quadratic    (int     a,       int  b,  int  c,
                   double* pRoot1, double* pRoot2);
void printResults (int     numRoots,
```

continued

PROGRAM 9-9 Quadratic Roots *(continued)*

```
                   int    a,     int  b, int  c,
                   double  root1, double  root2);

int main (void)
{
// Local Declarations
   int    a;
   int    b;
   int    c;
   int    numRoots;
   double root1;
   double root2;
   char   again = 'Y';

// Statements
   printf("Solve quadratic equations.\n\n");
   while (again == 'Y' || again == 'y')
      {
       getData (&a, &b, &c);
       numRoots = quadratic (a, b, c, &root1, &root2);
       printResults (numRoots, a, b, c, root1, root2);

       printf("\nDo you have another equation (Y/N): ");
       scanf (" %c", &again);
      } // while
   printf("\nThank you.\n");
   return 0;
}  // main

/* ================= getData =================
   Read coefficients for quadratic equation.
      Pre   a, b, and c contains addresses
      Post  data read into addresses in main
*/
void getData   (int* pa, int* pb, int* pc)
{
// Statements
   printf("Please enter coefficients a, b, & c: ");
   scanf ("%d%d%d", pa, pb, pc);

   return;
}  // getData

/* ================= quadratic =================
```

continued

PROGRAM 9-9 Quadratic Roots *(continued)*

```
   Compute the roots for a quadratic equation.
      Pre    a, b, & c are the coefficients
             pRoot1 & pRoot2 are variable pointers
      Post   roots computed, stored in calling function
      Return  2 two roots,
              1 one root,
              0 imaginary roots
             -1 not quadratic coefficients.
*/
int quadratic (int a, int b, int c,
               double* pRoot1, double* pRoot2)
{
// Local Declarations
   int result;

   double discriminant;
   double root;

// Statements
   if (a == 0 && b == 0)
      result = -1;
   else
      if (a == 0)
         {
          *pRoot1 = -c / (double) b;
          result = 1;
         } // a == 0
      else
         {
          discriminant =  b * b  - (4 * a * c);
          if (discriminant >= 0)
             {
              root     = sqrt(discriminant);
              *pRoot1 = (-b + root) / (2 * a);
              *pRoot2 = (-b - root) / (2 * a);
              result  = 2;
             } // if >= 0
          else
              result = 0;
         } // else
   return result;
}  // quadratic

/* ================ printResults ================
```

continued

PROGRAM 9-9 Quadratic Roots *(continued)*

```
   Prints the factors for the quadratic equation.
      Pre   numRoots contains 0, 1, 2
            a, b, c contains original coefficients
            root1 and root2 contains roots
      Post  roots have been printed
*/
void printResults  (int    numRoots,
                    int    a,  int b,  int c,
                    double root1, double root2)
{
// Statements
   printf("Your equation: %dx**2 + %dx + %d\n",
           a, b, c);
   switch (numRoots)
      {
       case 2:  printf("Roots are: %6.3f & %6.3f\n",
                        root1, root2);
                break;
       case 1:  printf("Only one root: %6.3f\n",
                        root1);
                break;
       case 0:  printf("Roots are imaginary.\n");
                break;
       default: printf("Invalid coefficients\n");
                break;
      } // switch
   return;
} // printResults
// ================== End of Program =================
```

```
Results:
Solve quadratic equations.

Please enter the coefficients a, b, & c: 2 4 2
Your equation: 2x**2 + 4x + 2
Roots are: -1.000 & -1.000

Do you have another equation (Y/N): y
Please enter the coefficients a, b, & c: 0 4 2
Your equation: 0x**2 + 4x + 2
Only one root: -0.500

Do you have another equation (Y/N): y
Please enter the coefficients a, b, & c: 2 2 2
Your equation: 2x**2 + 2x + 2
```

continued

PROGRAM 9-9 Quadratic Roots *(continued)*

```
Roots are imaginary.

Do you have another equation (Y/N): y
Please enter the coefficients a, b, & c: 0 0 2
Your equation: 0x**2 + 0x + 2
Invalid coefficients

Do you have another equation (Y/N): y
Please enter coefficients a, b, & c: 1 -5 6
Your equation: 1x**2 + -5x + 6
Roots are:  3.000 &  2.000

Do you have another equation (Y/N): n

Thank you.
```

Program 9-9 Analysis This problem has many interesting points. The function *main* is a test driver—that is, code that will not be put into production. Therefore, we code much of the test logic in *main* rather than providing separate functions for it.

The *scanf* function used in `getData` does not have the usual address operator (&) in front of variables `a`, `b`, and `c`. Why? Because they are already pointer values pointing to variables in *main* and therefore are already addresses.

The variables in these examples are either integers or pointers to integers. Parameters that receive something from *main* are integers whose values will be filled when the call is made. Those that send data back to *main* are pointers to integers that will be filled with the addresses of the corresponding variables in *main*. As a general rule, if a value will be changed, it must be passed as a pointer. If it will not be changed, it should be passed as a value. This protects data from accidental destruction.

In `quadratic`, note the extensive testing to make sure that the coefficients are valid. To ensure valid code, they are all necessary. Look at how we calculated the square root of the discriminant separately (statement 89). Since square root is a complex function, it is more efficient to call it just once and save the value for later use. Note also that the function has only one *return* statement. This is proper structured code, though many professional programmers would simply return at statements 77, 82, 92, and 95.

Study our test data carefully. Note that this set of test data executes every line of code in the program. Designing test data that completely validates a function is not an easy task. Ensuring that all code has been executed is even more difficult and tedious. One way to make sure that all code has been tested is to use your debugger to set a break point at every statement and then clear them as the program executes. When all break points have been cleared, you know every instruction has been executed.

Executing every line of code does not ensure that the function has no bugs, however. With large programs, it is virtually impossible to test every possible combination of data. One of the advantages of structured programming is that by breaking the program down into separate functions we can test it better.

This program has a potential problem. Do you see it? Hint: What if the user enters invalid data? There is no error checking in `getData`. If this were a production program, it would contain code to check for errors. It would then return a status flag to indicate if `getData` was successful or not.

9.6 Software Engineering

In the previous chapters, software engineering has always been related to the material in the chapter. This chapter is an exception. In this chapter, we discuss a general software engineering topic, quality, which can be applied to any topic, including pointers.

You will find no one who would even consider minimizing software quality, at least publicly. Everyone wants the best software available, and to listen to the creators of systems on the market, their systems are all perfect. Yet, as users of software, we often feel that quality software is a contradiction in terms. We all have our favorite software products, but not one of them is without a wart or two.

Since you are now moving into the world to be one of those software creators, you need to be aware of the basic concepts of software quality. In this chapter, we discuss some of the attributes of a quality product and how you go about achieving quality.

Quality Defined

Quality software is defined as

Software that satisfies the user's explicit and implicit requirements, is well documented, meets the operating standards of the organization, and runs efficiently on the hardware for which it was developed.

Every one of these attributes of good software falls squarely on you, the system designer and programmer. Note that we place on you the burden of satisfying not only the users' explicit requirements, but also their implicit needs. Often, users don't fully know what they need. When this happens, it is your job to determine their implicit requirements, which are hidden in the background. This is a formidable task indeed.

Of course, it is also your job to document the software. If you are lucky, you will have a technical writer to help, and even if you do, the final product is still your responsibility. And as an analyst and programmer, you are expected to know the standards of your organization and to implement them properly.

Finally, it is your program, so you are responsible for its efficiency. This means that you are expected to use appropriate and efficient algorithms. This was the focus of our discussions in Chapters 6 and 8 when we talked about analyzing algorithms and the big-O theory.

But quality software is not just a vague concept. If we want to attain it, we have to be able to measure it. Whenever possible, these measurements should be quantitative; they should be numerically measurable. For example, if an organization is serious about quality, it should be able to tell you the number of errors (bugs) per thousand lines of code and the mean time between failures for every software system it maintains. These are measurable statistics.

On the other hand, some of the measurements may be qualitative, meaning that they cannot be numerically measured. Flexibility and testability are examples of qualitative software measurements. This does not mean that they can't be measured, but rather that they rely on someone's judgment in assessing the quality of a system.

Quality Factors

Software quality can be divided into three broad measures: operability, maintainability, and transferability. These measures can be broken down as shown in Figure 9-26.

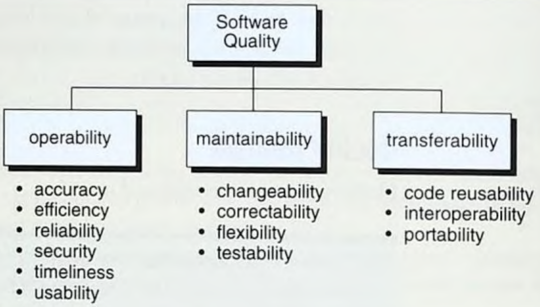


FIGURE 9-26 Software Quality

Operability

Operability refers to the basic operation of a system. The first thing a user notices about a system is its "look and feel." This means, especially for an online, interactive system, how easy and intuitive it is to use. Does it fit well into the operating system it is running under? For example, if it is running in a Windows environment, its pull-down and pop-up menus should work the same way the operating system's menus do. In short, operability answers the question, "How does it drive?"

But these factors are subjective; they are not measurable. So let's look at the factors that comprise operability. They are listed alphabetically.

Accuracy

A system that is not accurate is worse than no system at all. Most workers would rather rely on intuition and experience than a system that they know gives false and misleading information.

Any system that you develop, therefore, must be thoroughly tested, both by you (whitebox) and by a systems test engineer and the user (blackbox). If you get the opportunity, take a course on software testing.

Accuracy can be measured by such metrics as mean time between failures, number of bugs per thousand lines of code, and number of user requests for change.

Efficiency

Efficiency is, by and large, a subjective term. In some cases the user will specify a performance standard, such as that a real-time response must be received within 1 second, 95% of the time. This is certainly measurable.

Reliability

Reliability is really the sum of the other factors. If users count on the system to get their job done and are confident in it, then it is most likely reliable. On the other hand, some measures speak directly to a system's reliability, most notably, mean time between failures.

Security

The security of a system refers to how easy it is for unauthorized persons to access the system's data. Although this is a subjective area, there are checklists that assist in assessing the system's security. For example, does the system have and require passwords?

Timeliness

Does the system deliver its output in a timely fashion? For online systems, does the response time satisfy the users' requirements? For batch systems, are the reports delivered in a timely fashion? It is also possible, if the system has good auditability, to determine if the data in the system are timely; that is, are data recorded within a reasonable time after the activity that creates them takes place?

Usability

Usability is also highly subjective. The best measure of usability is to watch the users and see if they are using the system. User interviews will often reveal problems with the usability of a system.

Maintainability

Maintainability refers to keeping a system running correctly and up to date. Many systems require regular changes, not because they were poorly implemented but because of changes in external factors. For example, the payroll system for a company must be changed yearly, if not more often, to meet changes in government laws and regulations.

Changeability

The ease of changing a system is a subjective factor. Experienced project leaders, however, can estimate how long a requested change will take. If it takes too long, it may well be because the system is difficult to change. This is especially true of older systems.

Current software measurement tools estimate a program's complexity and structure. These tools should be used regularly, and if a program's complexity is high, we should consider rewriting the program. Programs that have been changed many times may have often lost their structured focus and become difficult to change. They also should be rewritten.

Correctability

One measure of correctability is mean time to recovery—how long it takes to get a program back in operation when it fails. Although this is a reactive definition, there are currently no predictors of how long it will take to correct a program before it fails.

Flexibility

Users are constantly requesting changes in systems. This qualitative attribute attempts to measure how easy it is to make these changes. If a program needs to be completely rewritten to effect a change, it is not flexible. Fortunately, this factor became less of a problem with the advent of structured programming.

Testability

You might think that testability is a highly subjective area, but a test engineer has a checklist of factors that can be used to assess a program's testability.

Transferability

Transferability refers to the ability to move data and/or a system from one platform to another and to reuse code. In many situations, it is not an important factor. On the other hand, if you are writing generalized software, it can be critical.

Code Reusability

If functions are written so that they can be reused in different programs and on different projects, then they are highly reusable. Good programmers build libraries of reusable functions that they can use when they need to solve a similar problem.

Interoperability

Interoperability addresses the capability of sending data to other systems. In today's highly integrated systems, interoperability is a desirable attribute. In fact, it has become so important that operating systems now support the ability to move data between applications, such as between a word processor and a spreadsheet.

Portability

Portability addresses the ability to move software from one hardware platform to another; for example, from a Macintosh to a Windows environment or from an IBM mainframe to a VAX environment.

The Quality Circle

The first and most important point to recognize is that quality must be designed into a system. It can't be added as an afterthought. It begins at Step 1, determining the user requirements, and continues throughout the life of the system. Since quality is a continuous concept that, like a circle, never ends, we refer to it as the quality circle.

There are six steps to developing quality software: quality tools, technical reviews, formal testing, change control, standards, and measurement and reporting. These steps are shown in Figure 9-27.

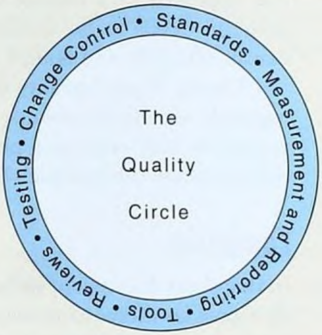

Change Control • Standards • Measurement and Reporting • Tools • Reviews • Testing •
The
Quality
Circle

FIGURE 9-27 The Quality Circle

While no one can deny that quality begins with the software engineers assigned to the team, they need quality tools to develop a quality product. Fortunately, today's development tools are excellent. A whole suite of quality tools known as computer-assisted software engineering (CASE) guides software development through requirements, design, programming and testing, and into production. Programmers can use workstations that not only assist in writing the program but also in testing and debugging. For example, some CASE tools track tests through a program and then determine which statements were executed and which were not. Tools such as these are invaluable for whitebox testing.

Another major step in quality software is the technical review. These reviews should be conducted at every step in the development process including requirements, design, programming, and testing. A typical program review begins after the programmer has designed the data structures and structure chart for a program. A design review board consisting of the systems analyst, test engineer, user representative, and one or two peers is then convened. Note that no one from management is allowed to attend a technical review. During the review, the programmer explains the approach and discusses interfaces to other programs while the reviewers ask questions and make suggestions.

Quality also requires formal testing. Formal testing ensures that the programs work together as a system and meet the defined requirements. After the programmer has completed unit testing, the program is turned over to another software engineer for integration and system testing. On a small project, this is most likely the systems analyst and/or the user. A large project will have a separate testing team.

Large systems take months and sometimes years to develop. It is only natural that, over extended periods of time, changes to the requirements and design become necessary. To ensure quality, each change should be reviewed and approved by a change control board. The impact of a change on each program needs to be assessed and properly planned. Uncontrolled change causes schedule and budget overruns and poor-quality products.

Finally, a good-quality environment measures all aspects of quality and regularly reports the results. Without measurement, you cannot tell if quality is good or bad, improving or deteriorating. At the same time, published standards provide the yardstick for many of the quality measurements.

Conclusion

In this short discussion, we have only introduced the concept of software quality. Consider these points as you design and program systems in the future. For a more complete discussion of the subject, we recommend Chapter 12 in Roger Pressman's *Software Engineering, A Practitioner's Approach.*[3]

[3] Roger Pressman, *Software Engineering, A Practitioner's Approach*. Sixth Edition (New York, N.Y., McGraw-Hill Book Company, 2005).

9.7 Tips and Common Programming Errors

1. The address of a memory location is a pointer constant and cannot be changed.
2. Only an address (pointer constant) can be stored in a pointer variable.
3. Remember compatibility. Do not store the address of a data variable of one type into a pointer variable of another type. In other words, a variable of pointer to *int* can only store the address of an *int* variable, and a variable of pointer to *char* can only store the address of a *char* variable.
4. The value of a data variable cannot be assigned to a pointer variable. In other words, the following code creates an error:

```
int* p;
int  a;
p = a;                                                  // ERROR
```

5. You must not use a pointer variable before it is assigned the address of a variable. In other words, the following lines create an error unless `p` is assigned an address:

```
int* p;
x    = *p;                                            // ERROR
*p   = x;                                             // ERROR
```

6. A pointer variable cannot be used to refer to a nonexistent pointed variable. For example, the following lines create a run-time error because `p` exists, but `*p` does not exist until it is assigned a value.

```
int* p;
*p = ...
```

7. You cannot dereference a pointer variable of type *void**.
8. Remember that the definition for a pointer variable allocates memory only for the pointer variable, not for the variable to which it is pointing.
9. The address operator (`&`) must be used only with an lvalue.
10. A function that uses addresses as parameters needs a pointer to data as a formal parameter; the actual parameter in the function call must be a pointer value (address).
11. If you want a called function to change the value of a variable in the calling function, you must pass the address of that variable to the called function.
12. When using multiple definitions in one statement—a practice we do not recommend—the pointer token is recognized only with one variable. Therefore, in the following definition, only the first variable is a pointer to an integer; the rest are integers.

```
int* ptrA, ptrB, ptrC;
```

13. It is a compile error to initialize a pointer to a numeric constant.

```
int* ptr = 59;
```

14. Similarly, it is a compile error to assign an address to any variable other than a pointer.

```
int x = &y;                                    // ERROR
```

15. It is a compile error to assign a pointer of one type to a pointer of another type without a cast. (Exception: If one pointer is a *void* pointer, it is permitted.)

16. It is a common compile error to pass values when addresses are required in actual parameters. Remember to use the address operator when passing identifiers to pointers in the called function.

17. It is a logic error to use a pointer before it has been initialized.

18. It is a logic error to dereference a pointer whose value is `NULL`.

9.8 Key Terms

indirection operator (*)
lvalue
pointer
pointer constant
pointer indirection
pointer variable
rvalue

9.9 Summary

❑ Every computer has addressable memory locations.

❑ A pointer is a derived data type consisting of addresses available in the computer.

❑ A pointer constant is an address that exists by itself. We cannot change it. We can only use it.

❑ The address operator (&) makes a pointer value from a pointer constant. To get the address of a variable, we simply use this operator in front of the variable name.

❑ The address operator can only be used in front of an lvalue. The result is an rvalue.

❑ A pointer variable is a variable that can store an address.

- ❑ The indirection operator (*) accesses a variable through a pointer containing its address.
- ❑ The indirection operator can only be used in front of a pointer value.
- ❑ Pointer variables can be initialized just like data variables. The initializer must be the address of a previously defined variable or another pointer variable.
- ❑ A pointer can be defined to point to other pointer variables (pointers to pointers).
- ❑ The value of a pointer variable can be stored in another variable if they are compatible—that is, if they are of the same type.
- ❑ One of the most useful applications of pointers is in functions.
- ❑ If we want a called function to access a variable in the calling function, we pass the address of that variable to the called function and use the indirection operator to access the variable.
- ❑ When we need to return more than one value from a function, we must use pointers.
- ❑ If a data item is not to be changed, it is passed to the called function by value. If a data item is to be changed, its address is passed to let the called function change its value.
- ❑ A function can also return a pointer value.
- ❑ In software engineering, *quality factors* refer to characteristics that a piece of software must have to become quality software.
- ❑ Quality factors are defined as operability, maintainability, and transferability.
- ❑ One of the most important points about the quality of a piece of software is that quality must be designed into the system. It cannot be added as an afterthought. To design quality into a system, we can use a tool called the quality circle.

9.10 Practice Sets

Review Questions

1. Pointer constants are drawn from the set of addresses for a computer.

   a. True
   b. False

2. Pointers are declared using the address operator as a token.

   a. True
   b. False

3. The value of a pointer type is always an address.
   - a. True
   - b. False

4. An rvalue expression can be used only to supply a value.
   - a. True
   - b. False

5. Which of the following statements about pointers is false?
   - a. Pointers are built on the standard type, address.
   - b. Pointers are machine addresses.
   - c. Pointers are derived data types.
   - d. Pointers can be defined variables.
   - e. Pointers can be constants.

6. The __________ operator is used with a pointer to dereference the address contained in the pointer.
   - a. address (`&`)
   - b. assignment (`=`)
   - c. indirection (`*`)
   - d. pointer (`^`)
   - e. selection (`->`)

7. Which of the following statements will not add 1 to a variable?
   - a. `a++;`
   - b. `a += 1;`
   - c. `a = a + 1;`
   - d. `*p = *p + 1;`
   - e. `*p++;`

8. Which of the following defines a pointer variable to an integer?
   - a. `int&  ptr;`
   - b. `int&& ptr;`
   - c. `int*  ptr;`
   - d. `int** ptr;`
   - e. `int^  ptr;`

9. Which of the following defines and initializes a pointer to the address of x?
   - a. `int* ptr = *x;`
   - b. `int* ptr = ^x;`
   - c. `int* ptr = &x;`
   - d. `int& ptr = *x;`
   - e. `int& ptr = ^x;`

10. *Pointers to pointers* is a term used to describe
    - a. any pointer that points to a variable.
    - b. any two pointers that point to the same variable.
    - c. any two pointers that point to variables of the same type.

d. pointers used as formal parameters in a function header.
e. pointers whose contents are the address of another pointer.

11. Given the following definitions,

```
int    x;
int*   p  = &x;
int**  pp = &p;
```

which answer can access the value stored in x?

a. p
b. pp
c. &p
d. *pp
e. **pp

12. Given the definitions shown below,

```
int     i;
float   f;
int*    pd;
float*  pf;
```

which answer is not valid?

a. `i     = 5;`
b. `f     = 5;`
c. `pd    = &i;`
d. `pf    = &f;`
e. `pd    = pf;`

13. Which of the following statements about pointer compatibility is true?

a. Because all pointers are addresses, pointers to different types can be assigned without a cast.
b. If pointers of different types are assigned, C uses an implicit cast.
c. When a pointer is cast, C automatically reformats the data to reflect the correct type.
d. When a *void* pointer is dereferenced, it must be cast.

14. Which of the following operators does not require an lvalue as its operand?

a. address operator (&total)
b. assignment (x = ...)
c. indirection (*ptr)
d. postfix increment (x++)
e. prefix increment (++x)

15. Which of the following is not an attribute of quality software?

a. Meets operating standards of organization.
b. Runs efficiently on any hardware.

c. Satisfies the user's explicit requirements.
d. Satisfies the user's implicit requirements.
e. Well documented.

Exercises

16. Declare and define the following:

a. A pointer variable `pi` pointing to an integer
b. A pointer variable `ppi` pointing to a pointer to an integer
c. A pointer variable `pf` pointing to a float
d. A pointer variable `ppc` pointing to a pointer to a char

17. If `a` is declared as integer, which of the following statements is true and which is false?

a. The expression `*&a` and `a` are the same.
b. The expression `*&a` and `&*a` are the same.

18. Given the following declarations:

```
int     x;
double  d;
int*    p;
double* q;
```

Which of the following expressions are not allowed?

a. `p = &x;`
b. `p = &d;`
c. `q = &x;`
d. `q = &d;`
e. `p = x;`

19. Given the following declarations:

```
int  a = 5;
int  b = 7;
int* p = &a;
int* q = &b;
```

What is the value of each of the following expressions?

a. `++a`
b. `++(*p)`
c. `--(*q)`
d. `--b`

20. What is the error (if any) in each of the following expressions?

a. `int  a = 5;`
b. `int* p = 5;`

c.
```
int  a;
int* p = &a;
```

d.
```
int   a;
int** q = &a;
```

21. Which of the following program segments is valid? Describe each error in the invalid statements.

a.
```
int* p;
scanf("%d", &p);
```

b.
```
int* p;
scanf("%d", &*p);
```

c.
```
int* p;
scanf("%d", *p);
```

d.
```
int  a;
int* p = &a;
scanf("%d", p);
```

22. Which of the following program segments has a logical error in it?

a.
```
int** p;
int*  q;
q = &p;
```

b.
```
int** p;
int*  q;
p = &q;
```

c.
```
int** p;
int** q;
p = &q;
```

d.
```
char   c = 'A';
char** p;
char*  q;
q = & c;
printf("%c", *p);
```

23. Given the following declaration:

```
int*** p;
```

What is the type of each of the following expressions?

a. `p`

b. `*p`

c. `**p`

d. `***p`

24. If `p` is a name of a variable, which of the following expressions is an lvalue and which one is an rvalue? Explain.

a. `p`

b. `*p`

c. `&p`

d. `*p + 2`

25. If `p` and `q` are pointer variable names and `a` is an array, which of the following expressions are not syntactically correct, because they violate the rules concerning lvalues and rvalues?

a. `*p = *p + 2;`

b. `&p = &a[0];`

c. `q = &(p + 2);`

d. `a[5] = 5;`

26. Write a function prototype statement for a function named `calc` that returns *void* and contains a reference parameter to *int*, `x`, and a reference parameter to a *long double*, `y`.

27. Write a function prototype statement for a function named `spin` that returns a pointer to an integer and contains a reference parameter to an *int*, `x`, and a pointer parameter to the address of a *long double*, `py`.

28. Assuming all variables are integers, and all pointers are typed appropriately, show the final values of the variables in Figure 9-28 after the following assignments.

```
a   = ***p;
s   = **p;
t   = *p;
b   = **r;
**q = b;
```

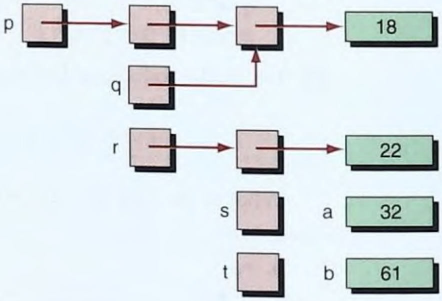


FIGURE 9-28 Exercise 28

29. Assuming all variables are integer, and all pointers are typed appropriately, show the final values of the variables in Figure 9-29 after the following assignments.

```
t   = **p;
b   = ***q;
*t  = c;
v   = r;
w   = *s;
a   = **v
*u  = *w
```

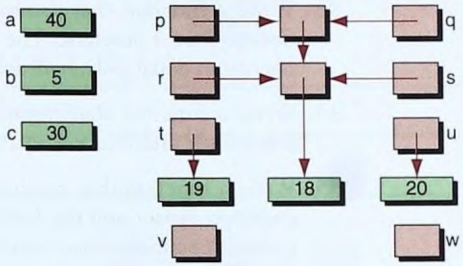


FIGURE 9-29 Exercise 29

30. In the following program, show the configuration of all the variables and the output.

```
#include <stdio.h>
int main (void)
{
// Local Declarations
   int   a;
   int*  p;
   int** q;
// Statements
   a = 14;
   p = &a;
   q = &p;
   printf("%d\n" ,   a);
   printf("%d\n ",  *p);
   printf("%d\n ", **q);
   printf("%p\t", p);
   printf("%p", q);
   return 0;
 } // main
```

Problems

31. Write a function that converts a Julian date to a month and day. A Julian date consists of a year and the day of the year relative to January 1. For

example, day 41 is February 10. The month and day are to be stored in integer variables whose addresses are passed as parameters. The function is to handle leap years. (For a discussion of the calculation of a leap year, see Chapter 5, Project 60.) If there is an error in a parameter, such as a day greater than 366, the function will return zero. Otherwise, it returns a positive number.

32. Modify Program 9-3 to include subtraction, multiplication, division, and remainder.

33. Write a function that receives a floating-point number representing the change from a purchase. The function will pass back the breakdown of the change in dollar bills, half-dollars, quarters, dimes, nickels, and pennies.

34. Write a function that receives a floating-point number and sends back the integer and fraction parts.

35. Write a function that receives two integers and passes back the greatest common divisor and the least common multiplier. The calculation of the greatest common divisor can be done using Euclid's method of repetitively dividing one number by the other and using the remainder (modulo). When the remainder is zero, the greatest common divisor has been found. For example, the greatest common divisor of 247 and 39 is 13, as in Table 9-4.

num1	num2	Remainder
247	39	13
39	(gcd) 13	0

TABLE 9-4 Greatest Common Divisors

Once you know the greatest common divisor (`gcd`), the least common multiplier (`lcm`) is determined as shown below.

$$\mathbf{lcm} = \frac{(\mathbf{num1} \times \mathbf{num2})}{\mathbf{gcd}}$$

Projects

36. Write a program that creates the structure shown in Figure 9-30 and then reads an integer into variable `a` and prints it using each pointer in turn. That is, the program must read an integer into variable `a` and print it using `p`, `q`, `r`, `s`, `t`, `u`, and `v`.

37. Write a program that creates the structure shown in Figure 9-31. It then reads data into `a`, `b`, and `c` using the pointers `p`, `q`, and `r`.

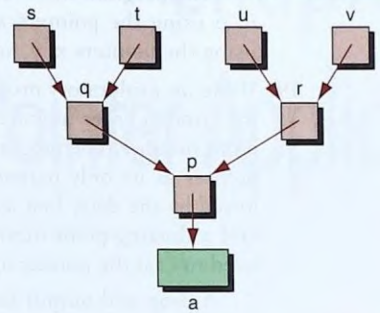


FIGURE 9-30 Structure for Project 36

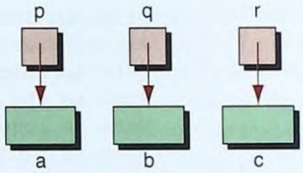


FIGURE 9-31 Data Structure for Project 37

After the data have been read, the program reassigns the pointers so that p points to c, q points to a, and r points to b. After making the reassignments, the program prints the variables using the pointers. For each variable, print both its contents and its address.

38. Write a program that creates the structure shown in Figure 9-32 and reads data into a and b using the pointers x and y.

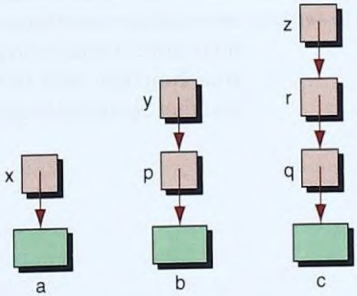


FIGURE 9-32 Data Structure for Project 38

The program then multiplies the value of a by b and stores the result in c using the pointers x, y, and z. Finally, it prints all three variables using the pointers x, y, and z.

39. Write an exploratory program to study pointers. In this program, prompt the user to enter a character, an integer, a long integer, and a floating-point number. The program then calls a print function that accepts a *void* pointer as its only parameter and prints the address of the pointer followed by the data, first as a character, then as an integer, a long integer, and a floating-point number. Since *void* pointers have no type, you will need to cast the pointer appropriately in each of your print statements.

A proposed output format is shown below. It is the result of calling the print function with a pointer to integer. You will most likely get different results. Call the print function for each of the user inputs, each time printing the results shown below. (The first line may be printed before calling the print function.)

```
Printing int 123
Printing data at location: 01D70C44
Data as char :
Data as int  :     123
Data as long : 8069185
Data as float:    0.00
```

Using the addresses that you printed, build a memory map of the variables as they existed in your program. Then, by analyzing the results of the printout, write a short explanation of what happened when your program ran. To fully understand the results, you may have to refer to your system documentation to determine the physical memory sizes of the various types.

40. Assume an environment in which transactions and responses are being sent across a network. When a transaction is sent, a time stamp is created and attached to the transaction. When the response is received, the time is noted. Using an integral time stamp, as shown in Program 9-8, write a function to determine the elapsed time in hours, minutes, and seconds between the transaction submittal and its response. Assume that no transaction takes longer than 24 hours. Create a test driver to validate your function. Test the function with at least five test cases, including one that spans midnight and one that is longer than 24 hours.

Chapter 10
Pointer Applications

In this chapter we explore two basic uses of pointers: with arrays and in dynamic memory.

We begin with the use of pointers in implementing arrays. This discussion includes the important topic of pointer arithmetic, which allows us to process data in an array by adding or subtracting from pointer addresses.

Following a discussion of passing arrays to functions, we discuss one of the most powerful aspects of C, dynamic memory. We examine two different approaches to dynamic memory, static allocation and dynamic allocation.

We conclude with two applications, sorting with pointers and ragged arrays.

Objectives

❑ To understand the relationship between arrays and pointers

❑ To understand the design and concepts behind pointer arithmetic

❑ To write programs using arrays and pointer arithmetic

❑ To better understand the design behind passing arrays to functions

❑ To understand the C implementation of dynamic memory

❑ To write programs using static and dynamic memory allocation

❑ To understand and implement ragged arrays (arrays of pointers)

10.1 Arrays and Pointers

The name of an array is a pointer constant to the first element. Because the array's name is a pointer constant, its value cannot be changed. Figure 10-1 shows an array with the array name as a pointer constant.

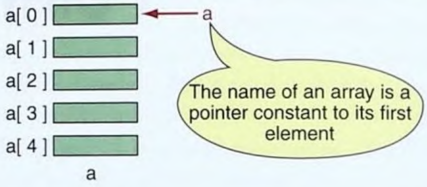

a[0]
a[1]
a[2]
a[3]
a[4]
a
a
The name of an array is a pointer constant to its first element

FIGURE 10-1 Pointers to Arrays

Since the array name is a pointer constant to the first element, the address of the first element and the name of the array both represent the same location in memory. We can, therefore, use the array name anywhere we can use a pointer, as long as it is being used as an rvalue. Specifically, this means that we can use it with the indirection operator. When we dereference an array name, we are dereferencing the first element of the array; that is, we are referring to `array[0]`. However, when the array name is dereferenced, it refers only to the first element, not the whole array.

a ←same→ &a[0]

a is a pointer only to the first element—not the whole array.

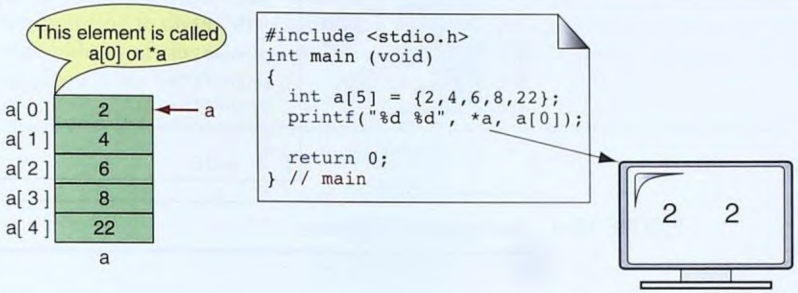
Prove this to yourself by writing a program with the code shown below. The code prints the address of the first element of the array (`&a[0]`) and the array name, which is a pointer constant. Note that we have used the format conversion code for a pointer (`%p`).

```
// Demonstrate that array name is pointer constant
   int a [5];
   printf("%p %p", &a[0], a);
```

We cannot tell you the values that this code will print, but they will be addresses in your computer. Furthermore, the first printed address (the first element in the array) and the second printed address (the array pointer) will be the same, proving our point.

The name of an array is a pointer constant; it cannot be used as an lvalue.

A simple variation on this code is to print the value in the first element of the array using both a pointer and an index. This code is demonstrated in Figure 10-2. Note that the same value, 2, is printed in both cases, again proving our point that the array name is a pointer constant to the beginning of the array.

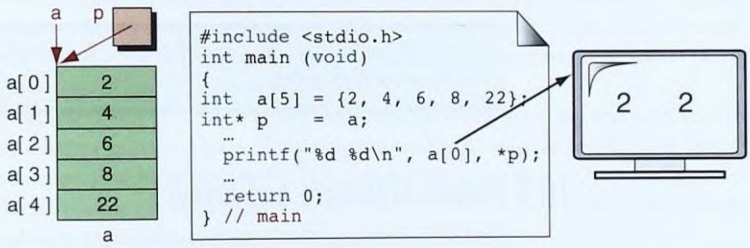


```
#include <stdio.h>
int main (void)
{
  int a[5] = {2,4,6,8,22};
  printf("%d %d", *a, a[0]);

  return 0;
} // main
```

```
2  2
```

FIGURE 10-2 Dereference of Array Name

Let's investigate another point. If the name of an array is really a pointer, let's see if we can store this pointer in a pointer variable and use it in the same way we use the name of the array. The program that demonstrates this is shown in Figure 10-3.



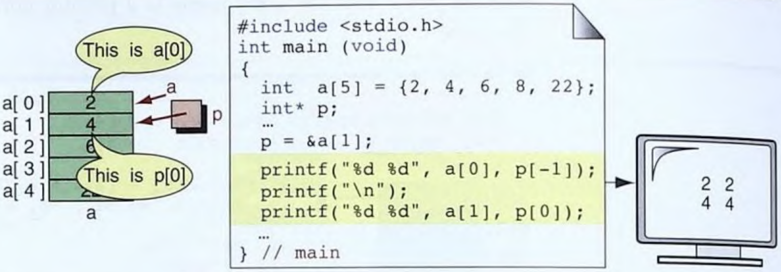
```
#include <stdio.h>
int main (void)
{
int  a[5] = {2, 4, 6, 8, 22};
int* p    =  a;
  …
  printf("%d %d\n", a[0], *p);
  …
  return 0;
} // main
```

```
2  2
```

FIGURE 10-3 Array Names as Pointers

Right after we define and initialize the array, we define a pointer and initialize it to point to the first element of the array by assigning the array name. Note especially that the array name is unqualified; that is, there is no address operator or index specification. We then print the first element in the array, first using an index notation and then pointer notation.

Let's look at another example that explores the close relationship between an array and a pointer. We store the address of the second element of the array in a pointer variable. Now we can use two different names to access each element. This does not mean that we have two arrays; rather, it shows that a single array can be accessed through different pointers.



```
#include <stdio.h>
int main (void)
{
  int  a[5] = {2, 4, 6, 8, 22};
  int* p;
  …
  p = &a[1];
  printf("%d %d", a[0], p[-1]);
  printf("\n");
  printf("%d %d", a[1], p[0]);
  …
} // main
```

```
2 2
4 4
```

FIGURE 10-4 Multiple Array Pointers

Figure 10-4 demonstrates the use of multiple names for an array to reference different locations at the same time. First, we have the array name. We then create a pointer to integer and set it to the second element of the array (`a[1]`). Now, even though it is a pointer, we can use it as an array name and index it to point to different elements in the array. We demonstrate this by printing the first two elements using first the array name and then the pointer. Note especially that, when a pointer is not referencing the first element of an array, the index can be negative `[-1]`. This is shown in the reference to `p[-1]`.

> **To access an array, any pointer to the first element can be used instead of the name of the array.**

10.2 Pointer Arithmetic and Arrays

Besides indexing, programmers use another powerful method of moving through an array: **pointer arithmetic**. Pointer arithmetic offers a restricted set of arithmetic operators for manipulating the addresses in pointers. It is especially powerful when we need to move through an array from element to element, such as when we are searching an array sequentially.

Pointers and One-Dimensional Arrays

If we have an array, `a`, then `a` is a constant pointing to the first element and `a + 1` is a constant pointing to the second element. Again, if we have a pointer, `p`, pointing to the second element of an array (see Figure 10-4), then `p - 1` is a pointer to the previous (first) element and `p + 1` is a pointer to the next (third) element. Furthermore, given `a`, `a + 2` is the address two elements from a, and a + 3 is the address three elements from a. We can generalize the notation as follows:

Given pointer, p, p ± n is a pointer to the value *n* elements away.

It does not matter how a and p are defined or initialized; as long as they are pointing to one of the elements of the array, we can add or subtract to get the address of the other elements of the array. This concept is portrayed in Figure 10-5.

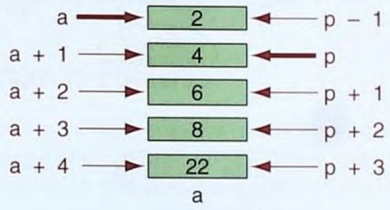


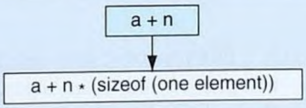
FIGURE 10-5 Pointer Arithmetic

But the meaning of adding or subtracting here is different from normal arithmetic. When we add an integer *n* to a pointer value, we get a value that corresponds to another index location, *n elements away*. In other words, *n* is an **offset** from the original pointer. To determine the new value, C must know the size of one element. The size of the element is determined by the type of the pointer. This is one of the prime reasons that pointers of different types cannot be assigned to each other.

If the offset is 1, then C can simply add or subtract one element size from the current pointer value. This may make the access more efficient than the corresponding index notation. If it is more than 1, then C must compute the offset by multiplying the offset by the size of one array element and adding it to the pointer value. This calculation is shown below.

address = pointer + (offset * size of element)

Depending on the hardware, the multiplication in this formula can make it less efficient than simply adding 1, and the efficiency advantage of pointer arithmetic over indexing may be lost.


a + n

a + n * (sizeof (one element))

We see the result of pointer arithmetic on different-sized elements in Figure 10-6. For *char*, which is usually implemented as 1 byte, adding 1 moves us to the next memory address (101). Assuming that integers are 4 bytes, adding 1 to the array pointer b moves us 4 bytes in memory (104). Finally, assuming the size of float is 6 bytes, adding 1 to array pointer c moves us 6 bytes in memory (106). In other words, a + 1 means different things in different situations.

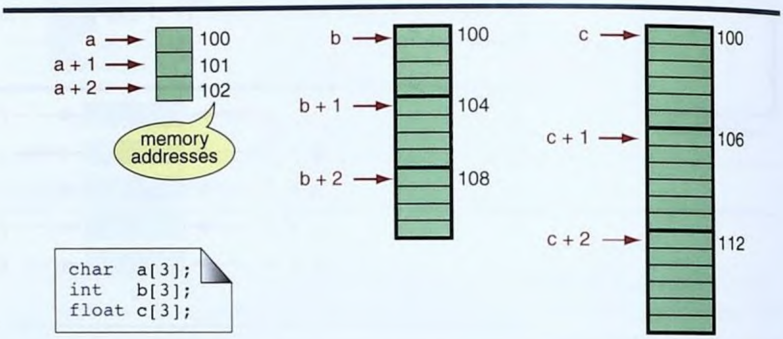


FIGURE 10-6 Pointer Arithmetic and Different Types

We've seen how to get the address of an array element using a pointer and an offset, now let's see how we can use that value. We have two choices: First, we can assign it to another pointer. This is a rather elementary operation that uses the assignment operator as shown below.

```
p = aryName + 5;
```

Second, we can use it with the indirection operator to access or change the value of the element we are pointing to. This possibility is seen in Figure 10-7.

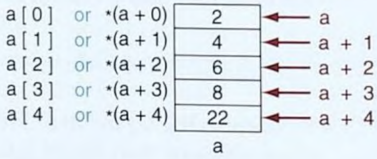


FIGURE 10-7 Dereferencing Array Pointers

For practice, let's use pointers to find the smallest number among five integers stored in an array. Figure 10-8 tracks the code as it works its way through the array.

The following expressions are identical.

`*(a + n)` and `a[n]`

We start with the smallest pointer (`pSm`) set to the first element of the array. The function's job is to see if any of the remaining elements are smaller. Since we know that the first element is not smaller than itself, we set the working pointer (`pWalk`) to the second element. The working pointer advances through the remaining elements, each time checking the element it is currently addressing against the smallest to that point (`pSm`). If the current element is smaller, its location is assigned to `pSm`.

Arithmetic Operations on Pointers

Arithmetic operations involving pointers are very limited. Addition can be used when one operand is a pointer and the other is an integer. Subtraction can be used only when both operands are pointers or when the first operand is a pointer and the second operand is an index integer. We can also manipulate a pointer with the postfix and unary increment and decrement operators. All of the following pointer arithmetic operations are valid:

```
p + 5    5 + p    p - 5    p1 - p2    p++    --p
```

When one pointer is subtracted from another, the result is an index representing the number of elements between the two pointers. Note, however, that the result is meaningful only if the two pointers are associated with the same array structure.

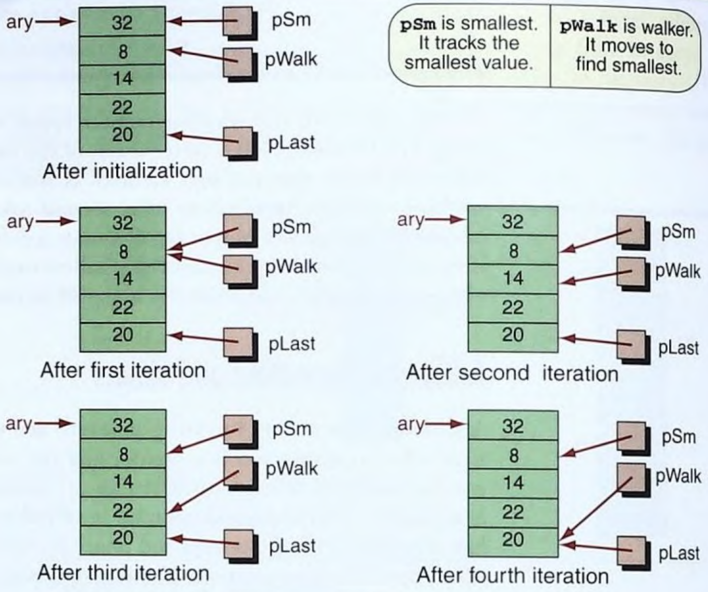
The relational operators (such as less than and equal) are allowed only if both operands are pointers of the same type. Two pointer relational expressions are shown below.

```
p1 >= p2        p1 != p2
```

The most common comparison is a pointer and the *NULL* constant, as shown in Table 10-1.

Long Form	Short Form
if (ptr == NULL)	if (!ptr)
if (ptr != NULL)	if (ptr)

TABLE 10-1 Pointers and Relational Operators



pSm is smallest. It tracks the smallest value.

pWalk is walker. It moves to find smallest.

```
pLast = ary + arySize - 1;
for (pSm = ary, pWalk = ary + 1;
     pWalk <= pLast;
     pWalk++)
   if (*pWalk < *pSm)
      pSm = pWalk;
```

FIGURE 10-8 Find Smallest

Using Pointer Arithmetic

In this section, we write two small programs that demonstrate moving through arrays using pointers.

Printing an Array

Program 10-1 prints an array, first by adding 1 to advance through the array and then subtracting 1 to print it backward.

PROGRAM 10-1 Print Array with Pointers

```
/* Print an array forward by adding 1 to a pointer. Then
   print it backward by subtracting one.
      Written by:
```

continued

PROGRAM 10-1 Print Array with Pointers *(continued)*

```
      Date:
*/
#include <stdio.h>

#define MAX_SIZE 10

int main (void)
{
// Local Declarations
   int  ary[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   int* pWalk;
   int* pEnd;

// Statements
   // Print array forward
   printf("Array forward : ");
   for (pWalk = ary, pEnd = ary + MAX_SIZE;
        pWalk <  pEnd;
        pWalk++)
      printf ("%3d", *pWalk);
   printf ("\n");

   // Print array backward
   printf ("Array backward: ");
   for (pWalk = pEnd - 1; pWalk >= ary; pWalk--)
      printf ("%3d", *pWalk);
   printf ("\n");

   return 0;
}  // main
```

```
Results:
Array forward :  1  2  3  4  5  6  7  8  9 10
Array backward: 10  9  8  7  6  5  4  3  2  1
```

Program 10-1 Analysis While this program is quite simple, it does demonstrate one efficiency point. Note that before we printed the forward loop, we set `pEnd`. Many programmers would use the following test in the limit condition:

pWalk < ary + MAX_SIZE

While this logic works, some compilers will calculate the end address in each loop, an obvious inefficiency. We recommend, therefore, that whenever possible you calculate the end pointer in the loop initialization.

Searching with Pointers

The logic in Program 10-1 works well for the sequential search but not for the binary search.[1] Recall that the binary search requires the calculation of the index or the address of the entry in the middle of a table. When we wrote the program using indexes, the calculation of the midpoint was done with the statement shown below.

```
mid = (first + last) / 2;
```

Since we cannot use addition with two pointers, this formula will not work. We need to come up with the pointer arithmetic equivalent. Another formula determines the midpoint in an array by calculating the number of elements from the beginning of the array. This method, known as the offset method, is shown below.

```
mid = first + (last - first) / 2;
```

The offset calculation works with pointers also. The subtraction of the first pointer from the last pointer will give us the number of elements in the array. The offset from the beginning of the array is determined by dividing the number of elements in the array by 2. We can then add the offset to the pointer for the beginning of the list to arrive at the midpoint. The pointer code is shown below.

```
midPtr = firstPtr + (lastPtr - firstPtr) / 2;
```

The pointer implementation of the binary search is shown in Program 10-2.

PROGRAM 10-2 Pointers and the Binary Search

```
/* ====================binary Search===================
   Search an ordered list using Binary Search
      Pre   list must contain at least one element
            endPtr is pointer to largest element in list
            target is value of element being sought
      Post  FOUND: locnPtr pointer to target element
                   return 1 (found)
          !FOUND: locnPtr = element below or above target
                   return 0 (not found)
*/
int binarySearch (int list[], int*  endPtr,
                  int target, int** locnPtr)
{
```

continued

[1] We discussed the binary search in Program 8-14 in Chapter 8.

PROGRAM 10-2 Pointers and the Binary Search *(continued)*

```
14   // Local Declarations
15      int* firstPtr;
16      int* midPtr;
17      int* lastPtr;
18
19   // Statements
20      firstPtr = list;
21      lastPtr  = endPtr;
22      while (firstPtr <= lastPtr)
23         {
24          midPtr = firstPtr + (lastPtr - firstPtr) / 2;
25          if (target > *midPtr)
26              // look in upper half
27              firstPtr = midPtr + 1;
28          else if (target < *midPtr)
29              // look in lower half
30              lastPtr = midPtr - 1;
31          else
32              // found equal: force exit
33              firstPtr = lastPtr + 1;
34         } // end while
35      *locnPtr = midPtr;
36      return (target == *midPtr);
37    }  // binarySearch
```

Program 10-2 Analysis Although the code in this function is relatively simple, the coding for `locnPtr` merits some discussion. In the calling function, `locnPtr` is a pointer to the found location. To store the pointer in `locnPtr`, therefore, we need to pass a pointer to a pointer to an integer (see statement 12). To create a pointer to pointer, the calling function must pass the address of its location pointer.

In addition to demonstrating the subtraction of two pointers, we also see the use of a relational operator with two pointers in statement 22.

Pointers and Two-Dimensional Arrays

The first thing to notice about two-dimensional arrays is that, just as in a one-dimensional array, the name of the array is a pointer constant to the first element of the array. In this case, however, the first element *is another array!* Assume that we have a two-dimensional array of integers. When we dereference the array name, we don't get one integer, we get an array of integers. In other words, the dereference of the array name of a two-dimensional array is a pointer to a one-dimensional array. Figure 10-9 contains a two-dimensional array and a code fragment to print the array.

Each element in the figure is shown in both index and pointer notation. Note that `table[0]` refers to an array of four integer values. The equivalent pointer notation is the dereference of the array name plus 0, `*(table + 0)`, which also refers to an array of four integers.

```
table[0] is identical to *(table + 0)
```

To demonstrate pointer manipulation with a two-dimensional array, let's print the table in Figure 10-9. To print the array requires nested *for* loops. When dealing with multidimensional arrays, however, there is no simple pointer notation. To refer to a row, we dereference the array pointer, which gives us a pointer to a row. Given a pointer to a row, to refer to an individual element, we dereference the row pointer. This double dereference is shown below.

```
*(*(table))
```

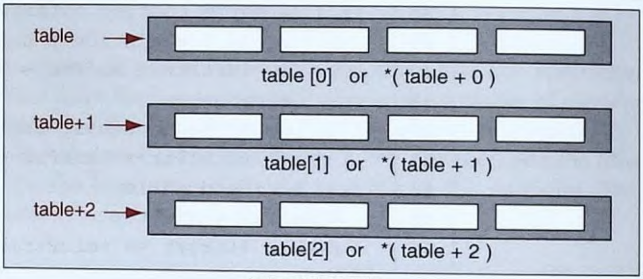


table
table [0] or *(table + 0)
table+1
table[1] or *(table + 1)
table+2
table[2] or *(table + 2)

int table[3][4] ;

```
for (i = 0; i < 3; i++)
  {
   for (j = 0; j < 4; j++)
      printf("%6d", *(*(table + i) + j));
   printf( "\n" );
  } // for i
```

Print Table

FIGURE 10-9 Pointers to Two-dimensional Arrays

But the previous expression refers only to the first element of the first row. To step through all the elements, we need to add two offsets, one for the row and one for the element within the row. We use loop counters, `i` and `j`, as the offsets. This is the same logic we saw when we printed a two-dimensional array using indexes. To print an element, we use the array name, `table`, and adjust it with the loop indexes. This gives us the relatively complex expression shown below.

```
*(*(table + i) + j)
```

This pointer notation is equivalent to the index syntax, `table[i][j]`. With multidimensional arrays, the pointer arithmetic has no efficiency advantage over indexing. Because the pointer notation for multidimensional arrays is so complex and there is no efficiency advantage, most programmers find it easier to use the index notation.

We recommend index notation for two-dimensional arrays.

10.3 Passing an Array to a Function

Now that we have discovered that the name of an array is actually a pointer to the first element, we can send the array name to a function for processing. When we pass the array, we do not use the address operator. Remember, the array name is a pointer constant, so the name is already the address of the first element in the array. A typical call would look like the following:

```
doIt (aryName);
```

The called program can declare the array in one of two ways. First, it can use the traditional array notation. This format has the advantage of telling the user very clearly that we are dealing with an array rather than a single pointer. This is an advantage from a structured programming and human engineering point of view.

```
int doIt (int ary[])
```

We can also declare the array in the function header as a simple pointer. The disadvantage to this format is that, while it is technically correct, it actually masks the data structure (array). For one-dimensional arrays, it is the code of choice among professional programmers.

```
int doIt (int* arySalary)
```

If you choose to code this way, use a good descriptive name for the parameter to minimize any reader confusion. The function documentation should also indicate clearly that an array is being passed.

Note, however, that when we pass a multidimensional array, we must use the array syntax in the header declaration and definition. The compiler needs to know the size of the dimensions after the first to calculate the offset for pointer arithmetic. Thus, to receive a three-dimensional array, we would use the following declaration in the function's header:

```
float doIt (int bigAry[][12][5])
```

To see how passing array pointers works, let's write a program that calls a function to multiply each element of a one-dimensional array by 2. The program's variables are shown in Figure 10-10, and the code is shown in Program 10-3.

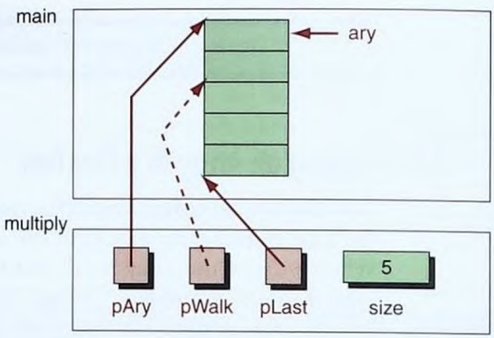


FIGURE 10-10 Variables for Multiply Array Elements By 2

PROGRAM 10-3 Multiply Array Elements by 2

```
/* Read from keyboard & print integers multiplied by 2.
      Written by:
      Date:
*/
#include <stdio.h>
#define SIZE  5

//  Function Declarations
void multiply (int* pAry, int size);

int main (void)
{
// Local Declarations
   int  ary [SIZE];
   int* pLast;
   int* pWalk;

// Statements
   pLast = ary + SIZE - 1;
   for (pWalk = ary; pWalk <= pLast; pWalk++)
      {
       printf("Please enter an integer: ");
```

continued

PROGRAM 10-3 Multiply Array Elements by 2 *(continued)*

```
       scanf ("%d", pWalk);
      } // for

   multiply (ary, SIZE);

   printf ("Doubled value is: \n");
   for (pWalk = ary; pWalk <= pLast; pWalk++)
        printf (" %3d", *pWalk);

   return 0;
}  // main

/* ==================== multiply ====================
   Multiply elements in an array by 2
      Pre   array has been filled
            size indicates number of elements in array
      Post  values in array doubled
*/
void multiply (int* pAry, int size)
{
// Local Declarations
   int* pWalk;
   int* pLast;

// Statements
   pLast = pAry + size - 1;
   for (pWalk = pAry; pWalk <= pLast; pWalk++)
       *pWalk = *pWalk * 2;
   return;
}  // multiply
// ================= End of Program =================
```

```
Results:
Please enter an integer: 1
Please enter an integer: 2
Please enter an integer: 3
Please enter an integer: 4
Please enter an integer: 5
Doubled value is:
   2   4   6   8  10
```

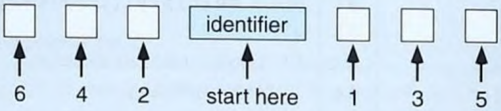
Program 10-3 Analysis This program contains several points of interest. First, we have declared the array in the function declaration using the more common pointer notation, but we have given it a name that indicates it is a pointer to an array (pAry).

In the multiply function, we use a separate pointer (pWalk) to walk through the list. We could have used pAry, since it was not being used other than to identify the beginning of the array. All too often, however, this type of "shortcut" saves a line or two of code only to create hours of debugging when the program is changed later. As we have pointed out before, do not use formal parameters as variables unless their intent is to change a value in the calling program. This rule is especially important when the parameter is a pointer, as in this case.

Finally, note that we have passed the size of the array to the multiply function. We still need to know how much data we need to process, and we use the size to calculate the address of the last element in the list. As a variation on the limit test, however, we could have passed a pointer to the last element of the array, &ary[SIZE - 1]. This would save the calculation of pLast. From a style and efficiency point of view, neither method has an advantage over the other. The structure and needs of other parts of the program usually dictate which method is used.

Understanding Complicated Declarations

To help you read and understand complicated declarations, we have developed the **right-left rule**. Using this rule to interpret a declaration, you start with the identifier in the center of a declaration and "read" the declaration by alternatively going right and then left until all entities have been read. The basic concept is shown below.



Consider the simple declaration

int x;

This is read as "x is ❒ an integer."[a]

```
int   x   ❒;
 ↑    ↑   ↑
 2    0   1
```

Since there is nothing on the right, we simply go left.

Now consider the example of a pointer declaration. This example is read as "p is ❒ a pointer ❒ to integer."

```
int   *   p   ❒   ❒;
 ↑    ↑   ↑   ↑   ↑
 4    2   0   1   3
```

Note that we keep going right even when there is nothing there until all the entities on the left have been exhausted. For a more complete discussion of complex declarations, see Appendix L.

[a] The box (❒) is just a place holder to show that there is no entry to be considered. Simply ignore it when you read each declaration.

10.4 Memory Allocation Functions

C gives us two choices when we want to reserve memory locations for an object: static allocation and dynamic allocation. Figure 10-11 shows the characteristics of memory allocation.

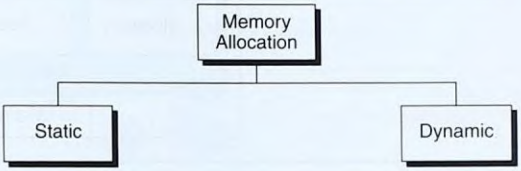


FIGURE 10-11 Memory Allocation

Memory Usage

To understand how dynamic memory allocation works, we must study how memory is used. Conceptually, memory is divided into program memory and data memory. Program memory consists of the memory used for *main* and all called functions. Data memory consists of permanent definitions, such as global data and constants, local declarations, and dynamic data memory. Exactly how C handles these different needs is a function of the operating system and the compiler writer's skills. We can, however, generalize the concepts.

Obviously, *main* must be in memory at all times. Beyond *main*, each called function must be in memory only while it or any of its called functions are active. As a practical matter, most systems keep all functions in memory while the program is running.

Although the program code for a function may be in memory at all times, the local variables for the function are available only when it is active. Furthermore, more than one version of the function can be active at a time. (See the discussion of recursion in Chapter 6.) In this case, multiple copies of the local variables are allocated, although only one copy of the function is present. The memory facility for these capabilities is known as the **stack memory**.

In addition to the stack, a memory allocation known as the heap is available. **Heap memory** is unused memory allocated to the program and available to be assigned during its execution. It is the memory pool from which memory is allocated when requested by the memory allocation functions. This conceptual view of memory is shown in Figure 10-12.

It is important to recognize that this is a conceptual view of memory. As we said before, implementation of memory is up to the software engineers who design the system. For example, nothing prevents the stack and the heap from sharing the same pool of memory. In fact, it would be a good design concept.

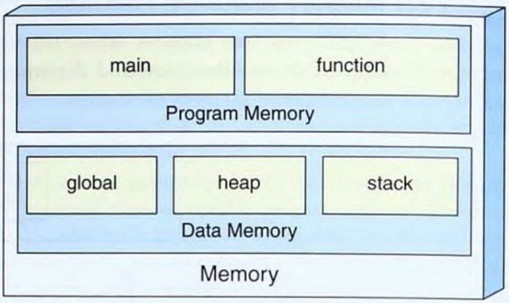


FIGURE 10-12 A Conceptual View of Memory

Static Memory Allocation

Static memory allocation requires that the declaration and definition of memory be fully specified in the source program. The number of bytes reserved cannot be changed during run time. This is the technique we have used to this point to define variables, arrays, pointers, and streams.

Dynamic Memory Allocation

Dynamic memory allocation uses predefined functions to allocate and release memory for data while the program is running. It effectively postpones the data definition, but not the data declaration, to run time.

We can refer to memory allocated in the heap only through a pointer.

To use dynamic memory allocation, we use either standard data types or derived types that we have previously declared. Unlike static memory allocation, dynamic memory allocation has no identifier associated with it; it has only an address that must be used to access it. To access data in dynamic memory, therefore, we must us a pointer. This concept is shown in Figure 10-13.

Memory Allocation Functions

Four memory management functions are used with dynamic memory. Three of them, *malloc*, *calloc*, and *realloc*, are used for memory allocation. The fourth, *free*, is used to return memory when it is no longer needed. All the memory management functions are found in the standard library file (*stdlib.h*). The collection of memory functions is shown in Figure 10-14.

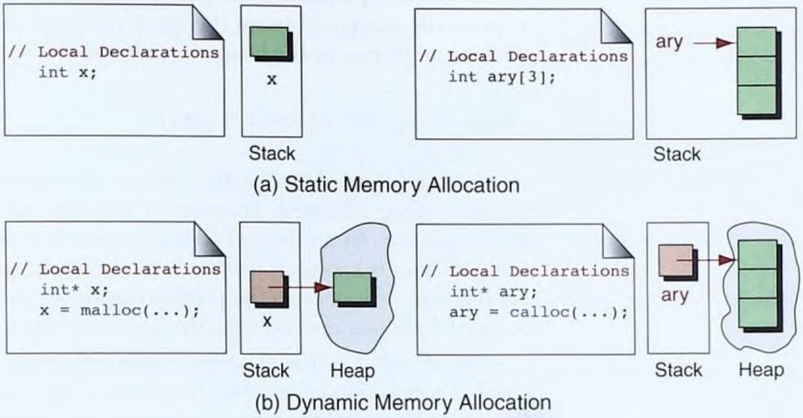


FIGURE 10-13 Accessing Dynamic Memory

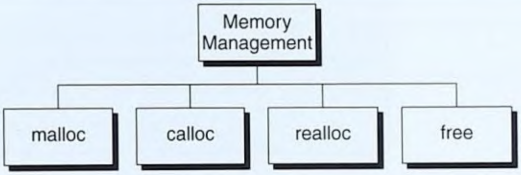


FIGURE 10-14 Memory Management Functions

Block Memory Allocation (malloc)

The *malloc* function allocates a block of memory that contains the number of bytes specified in its parameter. It returns a *void* pointer to the first byte of the allocated memory. The allocated memory is not initialized. We should therefore assume that it will contain unknown values and initialize it as required by our program.

The *malloc* function declaration is shown below.

```
void *malloc (size_t size);
```

The type, *size_t,* is defined in several header files including *stdio.h*. The type is usually an unsigned integer, and by the standard it is guaranteed to be large enough to hold the maximum address of the computer.

To provide portability, the size specification in *malloc*'s actual parameter is generally computed using the *sizeof* operator. For example, if we want to allocate an integer in the heap, we code the call as shown below.

```
pInt = malloc (sizeof (int));
```

As mentioned above, *malloc* returns the address of the first byte in the memory space allocated. However, if it is not successful, *malloc* returns a *NULL* pointer. An attempt to allocate memory from the heap when memory is insufficient is known as **overflow**. It is up to the program to check for memory overflow. If it doesn't, the program produces invalid results or aborts with an invalid address the first time the pointer is used.

Exactly what action should be taken when memory overflow is encountered is application dependent. If memory might be released by another portion of the program, the memory request can be held. Generally, however, the programmer must terminate the program and allocate more memory to the heap.

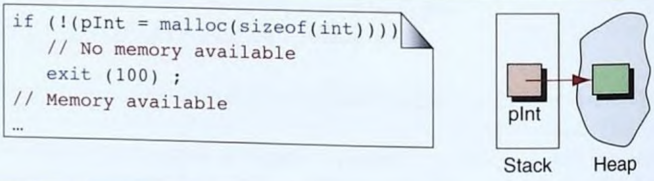
Memory Allocation Casting

Prior to C99, it was necessary to cast the pointer returned from a memory allocation function. While it is no longer necessary, it does no harm as long as the cast is correct. If you should be working with an earlier standard, the casting format is:

```
pointer = (type*) malloc(size)
```

The *malloc* function has one more potential error. If we call *malloc* with a zero size, the results are unpredictable. It may return a *NULL* pointer, or it may return some other implementation-dependent value. *Never call* malloc *with a zero size.*

Figure 10-15 shows a typical *malloc* call. In this example, we are allocating one integer object. If the memory is allocated successfully, `ptr` contains a value. If it doesn't, there is no memory and we exit the program with error code 100.



```
if (!(pInt = malloc(sizeof(int))))
   // No memory available
   exit (100) ;
// Memory available
...
```

FIGURE 10-15 *malloc*

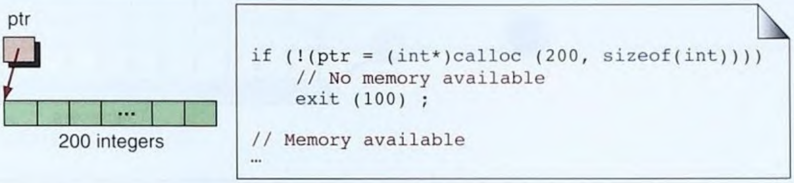
Contiguous Memory Allocation (calloc)

The second memory allocation function, *calloc*, is primarily used to allocate memory for arrays. It differs from *malloc* only in that it sets memory to null characters. The *calloc* function declaration is shown below.

```
void *calloc (size_t element-count,
              size_t element_size);
```

The result is the same for both *malloc* and *calloc* when overflow occurs and when a zero size is given.

A sample *calloc* call is shown in Figure 10-16. In this example, we allocate memory for an array of 200 integers.



```
if (!(ptr = (int*)calloc (200, sizeof(int))))
    // No memory available
    exit (100) ;

// Memory available
...
```

FIGURE 10-16 *calloc*

Reallocation Of Memory (*realloc*)

The *realloc* function can be highly inefficient and therefore should be used advisedly. When given a pointer to a previously allocated block of memory, *realloc* changes the size of the block by deleting or extending the memory at the end of the block. If the memory cannot be extended because of other allocations, *realloc* allocates a completely new block, copies the existing memory allocation to the new allocation, and deletes the old allocation. The programmer must ensure that any other pointers to the data are correctly changed. The operation of *realloc* is shown in Figure 10-17.

```
void* realloc (void* ptr, size_t newSize);
```

Releasing Memory (*free*)

When memory locations allocated by *malloc*, *calloc*, or *realloc* are no longer needed, they should be freed using the predefined function *free*. It is an error to *free* memory with a null pointer, a pointer to other than the first element of an allocated block, a pointer that is a different type than the pointer that allocated the memory; it is also a potential error to refer to memory after it has been released. The function declaration statement for *free* is shown below.

```
void free (void* ptr);
```

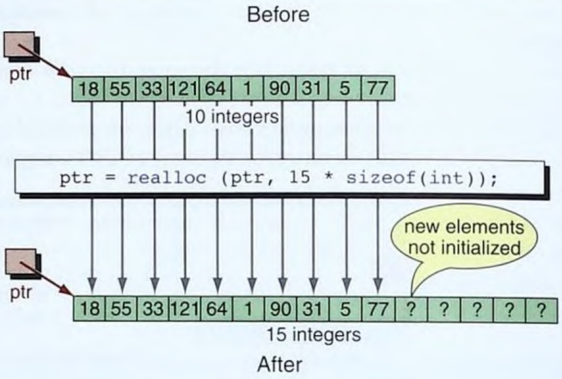


FIGURE 10-17 *realloc*

Figure 10-18 shows two examples. The first one releases a single element, allocated with a *malloc,* back to the heap. In the second example, the 200 elements were allocated with *calloc*. When we *free* the pointer in this case, all 200 elements are returned to the heap. You should note two things in this figure. First, it is not the pointers that are being released but rather what they point to. Second, to release an array of memory that was allocated by *calloc*, you need only release the pointer once. It is an error to attempt to release each element individually.

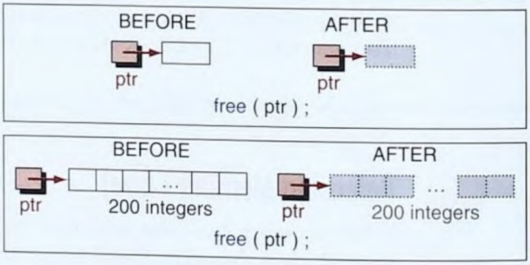


FIGURE 10-18 Freeing Memory

Releasing memory does not change the value in a pointer. It still contains the address in the heap. It is a *logic* error to use the pointer after memory has been released. The program may continue to run, but the data may be destroyed if the memory area is allocated for another use. This logic error is very difficult to trace. We suggest that immediately after you free memory you also clear the pointer by setting it to *NULL*.

Using a pointer after its memory has been released is a common programming error. Guard against it by clearing the pointer.

One final thought: You should free memory whenever it is no longer needed. It is not necessary, however, to clear memory at the end of the program. The operating system will release all memory when your program terminates.

The pointer used to free memory must be of the same type as the pointer used to allocate the memory.

10.5 Array of Pointers

Another useful structure that uses arrays and pointers is an **array of pointers**. This structure is especially helpful when the number of elements in the array is variable.

To look at an example, Table 10-2 is a two-dimensional array in which only one row (1) is full. The rest of the rows contain from one to four elements. This array is also known as a **ragged array** because the right elements in each row may be empty, giving it an uneven (ragged) right border.

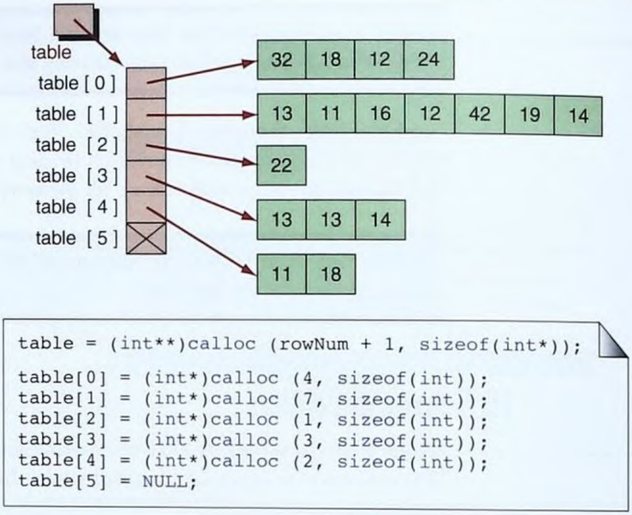
32	18	12	24			
13	11	16	12	42	19	14
22						
13	13	14				
11	18					

TABLE 10-2 A Ragged Table

If we use a two-dimensional array for storing these numbers, we are wasting a lot of memory. The solution in this case is to create five one-dimensional arrays that are joined through an array of pointers. One implementation of this concept is seen in Figure 10-19 along with the statements needed to allocate the arrays in the heap. Note that `table` is a pointer to a pointer to an integer and must be declared as shown below, not as an array.

```
int** table
```

Program 10-6 in Section 10.6, "Programming Applications," demonstrates the concept in a complete program. We will show other variations on this data structure in the next few chapters.



```
table = (int**)calloc (rowNum + 1, sizeof(int*));

table[0] = (int*)calloc (4, sizeof(int));
table[1] = (int*)calloc (7, sizeof(int));
table[2] = (int*)calloc (1, sizeof(int));
table[3] = (int*)calloc (3, sizeof(int));
table[4] = (int*)calloc (2, sizeof(int));
table[5] = NULL;
```

FIGURE 10-19 A Ragged Array

10.6 Programming Applications

This section contains two applications. The first is a rewrite of the selection sort, this time using pointers. The second uses dynamic arrays.

Selection Sort Revisited

Let's revisit the selection sort we developed in Chapter 8. Now that we know how to use pointers, we can improve it in several ways. First, and perhaps most important, it is structured. The structure chart is shown in Figure 10-20. Note that *main* contains no detailed code. It simply calls the three functions that will get the job done. First, `getData` reads data from the keyboard and puts it into an array. Then `selectSort` calls two functions to sort the data. Finally, `printData` displays the result. The complete algorithm is shown in Program 10-4.

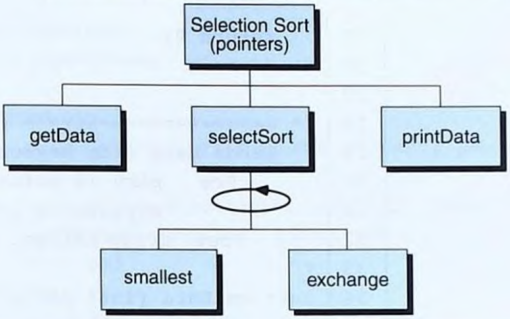


FIGURE 10-20 Selection Sort with Pointers—Structure Chart

PROGRAM 10-4 Selection Sort Revisited

```
/* Demonstrate pointers with Selection Sort
      Written by:
      Date written:
*/
#include <stdio.h>
#define SIZE 25

// Function Declarations
int*  getData    (int* pAry,    int  arySize);
void  selectSort (int* pAry,    int* last);
void  printData  (int* pAry,    int* last);
int*  smallest   (int* pAry,    int* pLast);
void  exchange   (int* current, int* smallest);

int main (void)
{
// Local Declarations
   int  ary[SIZE];
   int* pLast;

// Statements
   pLast = getData (ary, SIZE);
   selectSort (ary, pLast);
   printData  (ary, pLast);
```

continued

PROGRAM 10-4 Selection Sort Revisited *(continued)*

```
   return 0;
}  // main

/* ==================== getData ====================
   Reads data from keyboard into array for sorting.
      Pre   pAry is pointer to array to be filled
            arySize is integer with maximum array size
      Post  array filled. Returns address of last element
*/
int* getData (int* pAry, int arySize)
{
// Local Declarations
   int  ioResult;
   int  readCnt = 0;
   int* pFill   = pAry;

// Statements
   do
      {
       printf("Please enter number or <EOF>: ");
       ioResult = scanf("%d", pFill);
       if (ioResult == 1)
          {
           pFill++;
           readCnt++;
          } // if
      } while (ioResult == 1 && readCnt < arySize);

   printf("\n\n%d numbers read.", readCnt);
   return (--pFill);
}  // getData

/* ==================== selectSort ====================
   Sorts by selecting smallest element in unsorted
   portion of the array and exchanging it with element
   at the beginning of the unsorted list.
      Pre   array must contain at least one item
            pLast is pointer to last element in array
      Post  array rearranged smallest to largest
*/
void selectSort (int* pAry,  int* pLast)
{
// Local Declarations
```

continued

PROGRAM 10-4 Selection Sort Revisited *(continued)*

```c
   int* pWalker;
   int* pSmallest;

// Statements
   for (pWalker = pAry; pWalker < pLast; pWalker++)
       {
        pSmallest = smallest (pWalker, pLast);
        exchange (pWalker, pSmallest);
       } // for
   return;
} // selectSort

/* ==================== smallest ====================
   Find smallest element starting at current pointer.
      Pre   pAry points to first unsorted element
      Post  smallest element identified and returned
*/
int* smallest (int* pAry, int* pLast)
{
// Local Declarations
   int* pLooker;
   int* pSmallest;

// Statements
   for (pSmallest = pAry, pLooker = pAry + 1;
        pLooker <= pLast;
        pLooker++)
      if (*pLooker < *pSmallest)
          pSmallest = pLooker;
   return pSmallest;
} // smallest

/* ==================== exchange ====================
   Given pointers to two array elements, exchange them
      Pre   p1 & p2 are pointers to exchange values
      Post  exchange is completed
*/
void exchange (int* p1, int* p2)
{
// Local Declarations
   int temp;

// Statements
```

continued

PROGRAM 10-4 Selection Sort Revisited *(continued)*

```
   temp  =  *p1;
   *p1   =  *p2;
   *p2   =  temp;
   return;
}  // exchange

/* ==================== printData ====================
   Given a pointer to an array, print the data.
      Pre   pAry points to array to be filled
            pLast identifies last element in the array
      Post  data have been printed
*/
void  printData (int* pAry, int* pLast)
{
// Local Declarations
   int  nmbrPrt;
   int* pPrint;

// Statements
   printf("\n\nYour data sorted are: \n");
   for (pPrint = pAry, nmbrPrt = 0;
        pPrint <= pLast;
        nmbrPrt++, pPrint++)
      printf ("\n#%02d %4d", nmbrPrt, *pPrint);
   printf("\n\nEnd of List ");
   return;
}  // PrintData
// ================ End of Program =================
```

Program 10-4 Analysis Here are a few points you should note as you study this program. Note that we have used pointers and pointer arithmetic in all functions.

The `getData` function fills the array. Since the pointer, `pFill`, is always one ahead of the read, when we reach the end of file, it is pointing to an empty element. Therefore, when we return it we subtract 1.

The `selectSort` function advances through the array using a *for* statement. For each iteration, it selects the smallest element in the unsorted portion of the array and exchanges it with the first element in the unsorted portion of the array. Each loop, therefore, examines a smaller number of unordered elements. We stop at the element just before the last one because `smallest` always tests the first element and at least one element after the first one. When we are at the element just before the last, therefore, we are also testing the last element.

Finally, note the style used to code the *for* statement in both `smallest` and `printData`. Long *for* statements are more readable if you put each expression on a separate line.

Dynamic Array

This program creates a dynamic table that can store a ragged array. The column and the width of the **dynamic array** are tailored to the needs of the user. The program starts by asking the user for the number of rows that must be stored. After allocating the row pointers (using *calloc*), the program asks for the number of entries in each row. The table is then filled with data supplied by the user from the keyboard. To demonstrate the applications that could be used with this type of structure, we then determine the minimum, maximum, and average of each row of data. The design is shown in Figure 10-21.

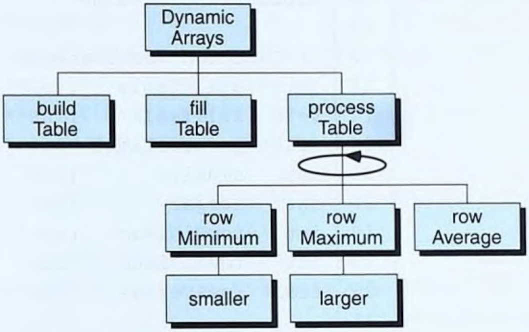


FIGURE 10-21 Dynamic Array Structure Chart

The data structure is shown in Figure 10-22. The table pointer points to the first pointer in an array of pointers. Each array pointer points to a second array of integers, the first element of which is the number of elements in the list. All arrays are allocated out of the heap, giving us a structure that is limited only by the computer's memory.

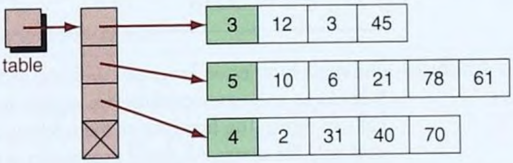


FIGURE 10-22 Ragged Array Structure

The complete set of programs to build and fill the table and some sample applications are shown in Programs 10-5 through 10-13. To compile these programs, they need to be put in one source file or combined using include statements.

PROGRAM 10-5 Dynamic Arrays: *main*

```
/* Demonstrate storing arrays in the heap. This program
   builds and manipulates a variable number of ragged
   arrays. It then calculates the minimum, maximum, and
   average of the numbers in the arrays.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// Function Declarations
int** buildTable   (void);
void  fillTable    (int** table);
void  processTable (int** table);
int   smaller      (int   first, int second);
int   larger       (int   first, int second);
int   rowMinimum   (int*  rowPtr);
int   rowMaximum   (int*  rowPtr);
float rowAverage   (int*  rowPtr);

int main (void)
{
// Local Declarations
   int** table;

// Statements
   table = buildTable();
   fillTable    (table);
   processTable (table);
   return 0;
}  // main
```

Program 10-5 Analysis The *main* function in Program 10-5 is a classic example of a well-designed program. It contains only one variable: the pointer to the array. There are only three functions calls in *main*: The first returns the address of the array; the other two use the array, and only the array. All detail processing is done in subfunctions.

PROGRAM 10-6 Dynamic Arrays: `buildTable`

```
/* ==================== buildTable ====================
   Create backbone of the table by creating an array of
   pointers, each pointing to an array of integers.
      Pre   nothing
```

continued

PROGRAM 10-6 Dynamic Arrays: **buildTable** *(continued)*

```
       Post  returns pointer to the table
*/
int** buildTable (void)
{
// Local Declarations
   int   rowNum;
   int   colNum;
   int** table;
   int   row;

// Statements
   printf("\nEnter the number of rows in the table: ");
   scanf ("%d", &rowNum);
   table = (int**) calloc(rowNum + 1, sizeof(int*));
   for (row = 0; row < rowNum; row++)
       {
        printf("Enter number of integers in row %d: ",
                row + 1);
        scanf ("%d", &colNum);
        table[row] = (int*)calloc(colNum + 1,
                                  sizeof(int));
        table[row] [0] = colNum;
       } // for
   table[row] = NULL;
   return table;
}  // buildTable
```

Program 10-6 Analysis We begin Program 10-6 by asking the user how many rows of data must be entered. Using the *calloc* function, we then allocate the memory for an array of pointers plus one extra pointer at the end. Each entry in the allocated table is used to point to an array of integers, also stored in the heap.

PROGRAM 10-7 Dynamic Arrays: **fillTable**

```
/* ==================== fillTable ====================
   This function fills the array rows with data.
      Pre   array of pointers
      Post  array filled
*/
void fillTable (int** table)
{
// Local Declarations
   int row = 0;

```

continued

PROGRAM 10-7 Dynamic Arrays: `fillTable` *(continued)*

```
// Statements
   printf("\n ==============================");
   printf("\n Now we fill the table.\n");
   printf("\n For each row enter the data");
   printf("\n and press return: ");
   printf("\n ==============================\n");

   while (table[row] != NULL)
      {
       printf("\n row %d (%d integers) =====> ",
               row + 1, table[row][0]);
       for (int column = 1;
                column <= *table[row];
                column++)
            scanf("%d", table[row] + column);
       row++;
      } // while
   return;
}  // fillTable
```

Program 10-7 Analysis Filling the rows, Program 10-7 requires a *while* statement to loop through the array pointers and a *for* statement to enter the data. We use the *while* statement because the pointer array is designed with a null pointer at the end, and we use it to tell we are at the end of the array. We use the *for* statement for filling the row because the user has already told us how many elements are in each row.

PROGRAM 10-8 Dynamic Arrays: Process Table

```
/* ================== processTable ====================
   Process the table to create the statistics.
      Pre   table
      Post  row statistics (min, max, and average)
*/
void processTable (int** table)
{
// Local Declarations
int   row = 0;
int   rowMin;
int   rowMax;
float rowAve;

// Statements
   while (table[row] != NULL)
      {
```

continued

PROGRAM 10-8 Dynamic Arrays: Process Table *(continued)*

```
      rowMin = rowMinimum (table[row]);
      rowMax = rowMaximum (table[row]);
      rowAve = rowAverage (table[row]);
      printf("\nThe statistics for row %d ", row + 1);
      printf("\nThe minimum: %5d",    rowMin);
      printf("\nThe maximum: %5d",    rowMax);
      printf("\nThe average: %8.2f ", rowAve);
      row++;
     } // while
   return;
}  // processTable
```

Program 10-8 Analysis Processing the table in Program 10-8 calls three functions to show how you could use a dynamic structure such as this. Obviously, many more applications could be used. Remember the structure of this example. You will be able to use this structure in future applications. Programs 10-9 through 10-13 continue the code for the functions.

PROGRAM 10-9 Dynamic Arrays: Find Row Minimum

```
/* =================== rowMinimum ====================
   Determines the minimum of the data in a row.
      Pre   given pointer to the row
      Post  returns the minimum for that row
*/
int rowMinimum (int* rowPtr)
{

// Local Declarations
   int rowMin = INT_MAX;

// Statements
   for (int column = 1; column <= *rowPtr; column++)
        rowMin = smaller (rowMin, *(rowPtr + column));
   return rowMin;
}  // rowMinimum
```

PROGRAM 10-10 Dynamic Arrays: Find Row Maximum

```
/* =================== rowMaximum ====================
   Calculates the maximum of the data in a row.
      Pre   given pointer to the row
      Post  returns the maximum for that row
*/
```

continued

PROGRAM 10-10 Dynamic Arrays: Find Row Maximum *(continued)*

```
int rowMaximum (int* rowPtr)
{
// Local Declarations
   int rowMax = INT_MIN;

// Statements
   for (int column = 1; column <= *rowPtr; column++)
        rowMax = larger  (rowMax, *(rowPtr + column));
   return rowMax;
}  // rowMaximum
```

PROGRAM 10-11 Dynamic Arrays: Find Row Average

```
/* ==================== rowAverage ====================
   This function calculates the average of data in a row.
      Pre   pointer to the row
      Post  returns the average for that row
*/
float  rowAverage (int* rowPtr)
{
// Local Declarations
   float  total = 0;
   float  rowAve;

// Statements
   for (int column = 1; column <= *rowPtr; column++)
        total += (float)*(rowPtr + column);
   rowAve = total / *rowPtr;
   return rowAve;
}  // rowAverage
```

PROGRAM 10-12 Dynamic Arrays: Find Smaller

```
/* ==================== smaller ====================
   This function returns the smaller of two numbers.
      Pre   two numbers
      Post  returns the smaller
*/
int  smaller (int first, int second)
{
// Statements
   return (first < second ? first : second);
}  // smaller
```

PROGRAM 10-13 Dynamic Arrays: Find Larger

```
/* ===================== larger =====================
   This function returns the larger of two numbers.
      Pre   two numbers
      Post  returns the larger
*/
int larger (int first, int second)
{
// Statements
   return (first > second ? first : second);
}  // larger
```

10.7 Software Engineering

Pointer applications need careful design to ensure that they work correctly and efficiently. The programmer not only must take great care in the program design but also must carefully consider the data structures that are inherent with pointer applications. The design of the data structures is beyond the scope of this text, but we can discuss the design of the pointers.

Before we discuss specific aspects of pointer applications, a word of caution: Remember the KISS principle. The complexity of pointers grows rapidly as you move from single references to double references to triple references. Keep it short and simple!

Pointers and Function Calls

You should always pass by value when possible. If you have to use a pointer to pass back a value, however, whenever possible, pass a pointer to the ultimate object to be referenced. When the pointer refers to the data variable, it is a single dereference. Despite your best efforts, at times you will have to pass a pointer to a pointer. When a function opens a file whose file pointer is in the calling function, you must pass a pointer to the pointer to the file table. In Figure 10-16 we allocated a dynamic array of 200 integers. If the allocation is performed in a subfunction, then it must receive a pointer to the pointer to the array in memory so that it can store the address of the array.

Whenever possible, use value parameters.

Pointers and Arrays

When you combine pointers and arrays, the complexity again quickly becomes difficult. This is especially true when the array is multidimensional. Whenever possible, therefore, rather than passing a multidimensional array, pass just one row. This reduces the complexity significantly, because the function is now dealing with a one-dimensional array. Not only are the references easier to work with, but passing a row allows simple pointer arithmetic, which is usually more efficient.

When you must work with a two-dimensional array, use index rather than pointer notation. Index notation is much simpler to work with, and there is no difference in efficiency. If you are not sure of this recommendation, consider the following equivalent expressions. Which one would you rather find in a strange program?

```
*(*(ary + i) + j)     or     a[i][j]
```

Array Index Commutativity

Commutativity is a principle in mathematics that says the results of an expression do not depend on the order in which the factors are evaluated. For example, a + b is identical to b + a. Pointer addition is commutative; subtraction is not. Thus, the following two expressions are identical.

```
a + i    i + a
```

But we also know that a + i is identical to a[i]. Similarly, i + a would be equivalent to i[a]. Therefore, using the principle of commutativity, we see that

a[i] is identical to i[a]

A word of caution. Commutavity works in C because of the pointer concept and pointer arithmetic. Do not try this in another language.

Dynamic Memory: Theory versus Practice

Do not get carried away with dynamic memory. The programming complexity of dynamically managing memory is very high. What you will often find, therefore, is that memory is not fully reused. To test memory reusability in your system, run Program 10-14. Note: This may not be a problem with today's compilers. When we reran this program for this edition, the memory was reused.

PROGRAM 10-14 Testing Memory Reuse

```
/* This program tests the reusability of dynamic memory.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
// Local Declarations
   int  looper;
   int* ptr;

// Statements
   for (looper = 0; looper < 5; looper++)
      {
       ptr = malloc(16);
       printf("Memory allocated at: %p\n", ptr);

```

continued

PROGRAM 10-14 Testing Memory Reuse *(continued)*

```
       free (ptr);
      } // for
   return 0;
}  // main
```

```
Results in Personal Computer:
Memory allocated at: 0x00e80238
Memory allocated at: 0x00e8024a
Memory allocated at: 0x00e8025c
Memory allocated at: 0x00e8026e
Memory allocated at: 0x00380280

Results in UNIX system:
Memory allocated at: 0x00300f70
Memory allocated at: 0x00300f70
Memory allocated at: 0x00300f70
Memory allocated at: 0x00300f70
Memory allocated at: 0x00300f70
```

Program 10-14 Analysis First look at the logic of this simple program. It loops five times, each time allocating 16 bytes and immediately freeing them. The program was run on two different systems, a personal computer and a UNIX network.

If the memory management is doing its job, the same 16 bytes would be allocated each time. What we see, however, is that for the personal computer, each address is different. It is also interesting to note that the difference between all the allocations except the first was 18 bytes, not 16.

The larger UNIX network system is more sophisticated, and it appears to reuse dynamic memory efficiently.

10.8 Tips and Common Programming Errors

1. If ary is an array, then

```
ary     is the same as     &ary[0]
```

2. If ary is the name of an array, then

```
ary[i]  is the same as  *(ary + i)
```

3. Remember that we usually pass the name of an array as a pointer value to a function that needs to access the elements of the array.

4. Remember that

```
int* a[5];  is different from  int (*a)[5];
```

5. Similar to array indexes, the most common pointer error is referencing a nonexistent element in an array. This is especially easy to do with pointer arithmetic.

6. It is a compile error to use pointer arithmetic with a pointer that does not reference an array.

7. It is a logic error to subtract two pointers that are referencing different arrays.

8. It is a compile error to subtract a pointer from an index.

9. It is a compile error to attempt to modify the name of an array using pointer arithmetic, such as

```
table++;                     // Error: table is constant
table = …;                   // Error: table is constant
```

10. The header file *stdlib.h* is required when using memory allocation functions.

11. It is a compile error to assign the return value from *malloc* or *calloc* to anything other than a pointer.

12. It is a logic error to set a pointer to the heap to *NULL* before the memory has been released.

13. It is a compile error to use pointer arithmetic with multiply, divide, or modulo operators.

10.9 Key Terms

array of pointers
dynamic array
dynamic memory allocation
offset
heap memory
overflow
pointer arithmetic
ragged array
right–left rule
stack memory
static memory allocation

10.10 Summary

- Arrays and pointers have a close relationship. The name of an array is a pointer constant to the first element of the array.
- The name of an array and the address of the first element in the array represent the same thing: an rvalue pointer.
- The name of an array is a pointer only to the first element, not the whole array.
- A pointer variable to the first element of an array can be used anywhere the name of the array is permitted, such as with an index.
- In pointer arithmetic, if `ptr` is pointing to a specific element in an array, `ptr + n` is the pointer value *n* elements away.
- The name of a two-dimensional array is a pointer to a one-dimensional array—the first row.
- In a multidimensional array, the following two expressions are equivalent.

```
*(*(a + i) + j)          a[i][j]
```

- We can pass an array to a function in many ways. One way is to pass the name of the array as a pointer.
- A ragged array—that is, an array of pointers—can be used to save space when not all rows of the array are full.
- The memory in a computer can be divided into program memory and data memory. Data memory can be partitioned into global area, heap, and stack.
- Static allocation of memory requires that the declaration and definition of memory be fully specified at compilation time.
- Dynamic allocation of memory is done during run time through the use of predefined functions.
- C has four predefined memory allocation functions: *malloc*, *calloc*, *realloc*, and *free*.
- To read and interpret a complex declaration, we can use the right–left rule.

10.11 Practice Sets

Review Questions

1. Given a pointer to an array element, `ptr`, `ptr - 5` is a pointer to the value 5 elements toward the beginning of the array.

   a. True
   b. False

2. Adding 1 to a pointer increases the address stored in it by 1 byte.

   a. True
   b. False

3. The parameter declaration `int* a` can be used to declare an array of integers passed to a function.

   a. True
   b. False

4. Dynamically allocated memory can only be referred to through pointers.

   a. True
   b. False

5. Which of the following statements about pointers and arrays is true?

   a. The following expressions are identical when `ary` is an array: `*ary` and `&ary[0]`.
   b. The following expressions are identical when `ary` is an array: `*ary` and `*ary[0]`.
   c. The name of an array can be used with the indirection operator to reference data.
   d. The name of the array is a pointer variable.
   e. The only way to reference data in an array is with the index operator.

6. Which of the following statements about pointer arithmetic is true?

   a. Any arithmetic operator can be used to change the value of a pointer.
   b. Given a pointer `ptr`, `ptr + n` is a pointer to the value *n* elements away.
   c. Pointer arithmetic is a short-hand notation that changes the value that a pointer is referencing.
   d. Pointer arithmetic is valid only with pointers to arithmetic variables, such as pointer to integer.
   e. Pointer arithmetic is valid only with the name of the array.

7. Which of the following parameters is a two-dimensional array of integers?

   a. `int ary`
   b. `int ary[ ][SIZE2]`
   c. `int* ary`
   d. `int* ary[ ][SIZE2]`
   e. `int* ary[SIZE2]`

8. Which of the following is not a C memory allocation function?
   - a. *alloc( )*
   - b. *calloc( )*
   - c. *free( )*
   - d. *malloc( )*
   - e. *realloc( )*

9. Which of the following statements about memory allocation is true?
   - a. Allocated memory can be referred to only through pointers; it does not have its own identifier.
   - b. *calloc* (change allocation) is used to change the allocation to memory previously allocated through *malloc*.
   - c. *malloc* (memory allocation) is used to allocate blocks of memory for arrays.
   - d. *realloc* (release allocation) is used to release memory when it is no longer needed.
   - e. Underflow can occur only with *calloc* and *malloc*.

10. The function that returns memory to the heap is:
    - a. *alloc( )*
    - b. *calloc( )*
    - c. *free( )*
    - d. *malloc( )*
    - e. *realloc( )*

11. Which of the following statements about releasing memory allocation is false?
    - a. It is an error to dereference a pointer to allocated memory after the memory has been released.
    - b. It is an error to *free* memory with a pointer to other than the first element of an allocated array.
    - c. Memory should be freed as soon as it is no longer needed.
    - d. Only one call to *free* is necessary to release an entire array allocated with *calloc*.
    - e. To ensure that it is released, allocated memory should be freed before the program ends.

12. Which of the following statements about ragged arrays is false?
    - a. Ragged arrays are two-dimensional arrays in which the right elements of a row may be empty.
    - b. Ragged arrays can be implemented using an array of pointers to one-dimensional arrays.
    - c. Ragged arrays can only be used with arrays of integers.
    - d. Ragged arrays implemented as an array of pointers save memory.
    - e. Ragged arrays implemented as an array of pointers can be created in dynamic memory.

Exercises

13. Rewrite each of the following expressions by replacing the index operator (`[...]`) with the indirection operator (`*`).

    a. `tax[6]`

    b. `score[7]`

    c. `num[4]`

    d. `prices[9]`

14. Rewrite each of the following expressions by replacing the indirection operator (`*`) with the index operator (`[...]`). Each identifier refers to an array.

    a. `*(tax + 4)`

    b. `*(score + 2)`

    c. `*(number + 0)`

    d. `*prices`

15. Imagine you have the following declarations:

```
int  ary[10];
int* p = &ary[3];
```

Show how you can access the sixth element of `ary` using the pointer `p`.

16. Given the following declarations:

```
float ary[200];
```

Write the function declaration for a function named `fun` that can manipulate a one-dimensional array of floating-point numbers, `ary`. Provide an additional parameter, a pointer to the last element in the array. Code a call to the function.

17. Show what would be printed from the following block:

```
{
 int  num[5] = { 3, 4, 6, 2, 1 };
 int* p = num;
 int* q = num +2;
 int* r = &num[1];
 printf("\n%d %d", num[2], *(num + 2));
 printf("\n%d %d", *p, *(p + 1));
 printf("\n%d %d", *q, *(q + 1));
 printf("\n%d %d", *r, *(r + 1));
} // Block
```

18. Show what would be printed from the following block.

```
// Function Declaration
   void printOne   (int*);
   void printTwo   (int*);
   void printThree (int*);
{
// Local Declarations
   int num [5] = {3 , 4 , 6 , 2 , 1};
// Statements
   printOne   (num);
   printTwo   (num + 2);
   printThree (&num [2]);
   return 0;
} // Block
void printOne (int* x)
{
   printf("\n%d", x[2]);
   return;
}  // printOne
void printTwo (int* x)
{
   printf("\n%d", x[2]);
   return;
}  // printTwo
void printThree (int* x)
{
   printf("\n%d", *x);
   return;
}  // printThree
```

19. Given the following definition:

```
int table [4][5];
```

write the function declaration for a function named `fun` that can accept the whole array using a pointer. Write the statement that calls this function.

20. Given the following definition:

```
int table [4][5];
```

write the function declaration for a function named `torture` that accepts one row of an array at a time.

21. Draw pictures to show the memory configuration for each of the following declarations.

a. `int* x [5];`

b. `int (*x) [5];`

22. Show what would be printed from the following block:

```
{
// Local Declarations
   int x [2][3] =
     {
      { 4  , 5 , 2 } ,
      { 7 ,  6 , 9 }
     };
   int (*p) [3]  = &x [1];
   int (*q) [3]  =  x;
// Statements
   printf("\n%d %d %d", (*p)[0],(*p)[1],(*p)[2]);
   printf("\n%d %d",     *q[0],  *q[1]);
}  // Block
```

23. Show what would be printed from the following block:

```
{
// Local Declarations
   int x [2][3] = {
                    { 4  , 5 , 2 } ,
                    { 7 ,  6 , 9 }
                  };
// Statements
   fun (x);
   fun (x + 1);
   return 0;
}  // Block
void fun (int (*p)[3])
{
 printf("\n%d %d %d", (*p)[0],(*p)[1],(*p)[2]);
 return;
}
```

24. Given the following definitions:

```
int  num[26] = {23, 3, 5, 7, 4, -1, 6};
int* n = num;
int  i = 2;
int  j = 4;
```

show the value of the following expressions:

a. `n`

b. `*n`

c. `*n + 1`

d. `*(n + 1)`

e. `*n + j`

f. `*&i`

25. Given the following definitions:

```
char  a[20] = {'z', 'x', 'm', 's', 'e', 'h'};
char* pa = a;
int   i  = 2;
int   j  = 4;
int*  pi = &i;
```

show the value of the following expressions:

a. `*(pa + j)`
b. `*(pa + *pi)`

26. Given the following definition:

```
int data[15] = {5,2,3,4,1,3,7,2,4,3,2,9,12};
```

show the value of the following expressions:

a. `data + 4`
b. `*(data + 4)`
c. `*data + 4`
d. `*(data + (*data + 2))`

27. Given the following definitions:

```
int  i  = 2;
int  j  = 4;
int* pi = &i;
int* pj = &j;
```

show the value of the following expressions:

a. `*&j`
b. `*&*&j`
c. `*&pi`
d. `**&pj`
e. `&**&pi`

28. Given the following definitions:

```
char a[20] = {'z', 'x', 'm', 's', 'e', 'h'};
int  i = 2;
int  j = 4;
```

write pointer expressions that evaluate to the same value as each of the following:

a. `a[0]`
b. `a[5]`
c. The address of the element just before `a[0]`.

d. The address of the last element in a.

e. The address of the element just after the last element in a.

f. The next element after a[3].

g. The next element after a[12].

h. The next element after a[j].

29. Given the following definitions:

```
int num[10] = {23,3,5,7,4,-1,6,12,10,-23};
int i = 2;
int j = 4;
```

write index expressions that evaluate to the same value as each of the following:

a. `*(num + 2)`

b. `*(num + i + j)`

c. `*(num + *(num + 1))`

d. `*(num + j)`

e. `*(num + i) + *(num + j)`

30. Given the following definition:

```
int num[2000] = {23, 3, 5, 7, 4, -1, 6};
```

write two pointer expressions for the address of num[0].

31. Given the following definitions:

```
int  num[26] = {23, 3, 5, 7, 4, -1, 6};
int  i = 2;
int  j = 4;
int* n = num;
```

write the equivalent expressions in index notation.

a. `n`

b. `*n`

c. `*n + 1`

d. `*(n + 1)`

e. `*(n + j)`

32. Given the following definitions:

```
int  num[26] = {23, 3, 5, 7, 4, -1, 6};
int* pn;
```

write a test to check whether pn points beyond the end of num.

33. Given the following function for mushem and the definitions shown below

```
int  mushem (int*, int*);
int  i = 2;
int  j = 4;
int* pi = &i;
int* pj = &j;
```

indicate whether each of the following calls to mushem is valid:

a. `i = mushem (2, 10);`

b. `j = mushem (&i, &j);`

c. `i = mushem (pi, &j);`

d. `j = mushem (i, j);`

e. `mushem (pi, pj);`

34. What is the output from the following program?

```
#include <stdio.h>
int fun (int*, int, int*);
int main (void)
{
// Local Declarations
   int a = 4;
   int b = 17;
   int c[5] = {9, 14, 3, 15, 6};
// Statements
   a = fun(&a, b, c);
   printf("2. %d %d %d %d %d %d %d\n",
            a, b, c[0], c[1], c[2], c[3], c[4]);
   return 0;
}  // main
   int fun (int* px, int y, int* pz)
{
// Local Declarations
   int  a = 5;
   int* p;
// Statements
   printf("1. %d %d %d\n", *px, y, *pz);
   for (p = pz; p < pz + 5; ++p)
        *p = a + *p;
   return (*px + *pz + y);
}  // fun
```

35. What is the output from the following program?

```
#include <stdio.h>
int sun (int*, int, int*);
```

continued

```
int main (void)
{
// Local Declarations
   int  a = 4;
   int  b = 17;
   int  c[5] = {9, 14, 3, 15, 6};
   int* pc = c;
// Statements
   a = sun(pc, a, &b);
   printf("2. %d %d %d %d %d %d %d\n",
            a, b, c[0], c[1], c[2], c[3], c[4]);
   return 0;
} // main
int sun (int* px, int y, int* pz)
{
// Local Declarations
   int  i = 5;
   int* p;
// Statements
   printf("1. %d %d %d\n", *px, y, *pz);
   for (p = px; p < px + 5; p++)
       *p = y + *p;
   *px = 2 * i;
   return (*pz + *px + y);
} // sun
```

Problems

36. We have two arrays, `A` and `B`, each containing 10 integers. Write a function that checks if every element of array `A` is equal to its corresponding element in array `B`. In other words, the function must check if `A[0]` is equal to `B[0]`, `A[1]` is equal to `B[1]`, and so on. The function must accept only two pointer values and return a Boolean, *true* for equal and *false* for unequal.

37. Generalize the function in Problem 36 to include the number of elements to be compared as a parameter.

38. Write a function that reverses the elements of an array in place. In other words, the last element must become the first, the second from last must become the second, and so on. The function must accept only one pointer value and return *void*.

39. The Pascal triangle can be used to compute the coefficients of the terms in the expansion of $(a + b)^n$. Write a function that creates a ragged array representing the Pascal triangle. In a Pascal triangle, each element is the sum of the element directly above it and the element to the left of the element directly above it (if any). A Pascal triangle of size 7 is shown in the following.

1						
1	1					
1	2	1				
1	3	3	1			
1	4	6	4	1		
1	5	10	10	5	1	
1	6	15	20	15	6	1

Your function must be able to create the triangle of any size. The function should accept an integer representing the size of the triangle and return a pointer to the array it created.

40. Write a function that tests an International Standard Book Number (ISBN) to see if it is valid. The ISBN is used to define a book uniquely. It is made of 10 digits, as shown below. For an ISBN to be valid, the weighted sum of the 10 digits must be evenly divisible by 11. The 10th digit may be X, which indicates 10. (If you are not familiar with the algorithm for the weighted sum, it is explained in Chapter 8, Problem 31). The ISBN format is shown in Figure 10-23.

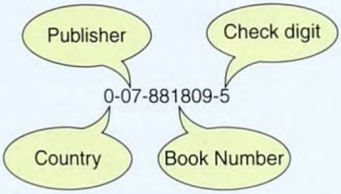


FIGURE 10-23 Project 40 Structure

The function must accept a pointer value (the name of the array) and return a Boolean, *true* for valid and *false* for invalid.

41. Write a function that copies a one-dimensional array of *n* elements into a two-dimensional array of *j* rows and *k* columns. The resulting array will be placed in the heap. The data will be inserted into the array in row order; that is, the first *k* items will be placed in row 0, the second *k* items in row 1, and so forth until all rows have been filled.

If *j* and *k* are not factors of *n*—that is, if $n \neq j * k$—the function returns a null pointer. Otherwise, it returns the pointer to the two-dimensional array. The input array and *j*, *k*, and *n* will be passed as parameters.

42. Given the following definition:

```
int num[20];
```

and using only pointer notation, write a *for* loop to read integer values from the keyboard to fill the array.

43. Given the following definition

```
char a[40];
```

and using only pointer notation, write a *for* loop to read characters from the keyboard to fill the array.

44. Given the following declaration and definition:

```
char a[6] = {'z', 'x', 'm', 's', 'e', 'h'};
```

and using only pointer notation, write a loop to rotate all values in a to the right (toward the end) by one element.

45. Repeat Problem 44, with the rotation one element to the left.

46. Write a function named addem with two call-by-address integer parameters. The function will add 2 to the first parameter and 5 to the second parameter. Test the function by calling it with the values of 17 and 25.

Projects

47. Write a program that reads integers from the keyboard and place them in an array. The program then will sort the array into ascending and descending order and print the sorted lists. The program must not change the original array or create any other integer arrays.

The solution to this problem requires two pointer arrays as shown in the example in Figure 10-24. The first pointer array is rearranged so that it points to the data in ascending sequence. The second pointer array is rearranged so that it points to the data in descending sequence.

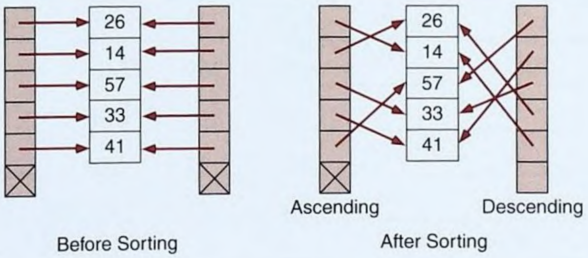


FIGURE 10-24 Sample Structure for Project 47

Your output should be formatted with the three arrays printed as a vertical list next to each other, as shown in Table 10-3.

Ascending	Original	Descending
14	26	57
26	14	41
33	57	33
41	33	26
57	41	14

TABLE 10-3 Format for Project 47

48. Write a program that creates a two-dimensional array in the heap and then analyzes it to determine the minimum, maximum, and average of each column.

The data are to be read from a file. The first two elements are the number of rows in the array and the number of columns in each row. The file data for a 12 × 8 array are shown below.

```
 12    8
838  758  113  515   51  627   10  419
212   86  749  767   84   60  225  543
 89  183  137  566  966  978  495  311
367   54   31  145  882  736  524  505
394  102  851   67  754  653  561   96
628  188   85  143  967  406  165  403
562  834  353  920  444  803  962  318
422  327  457  945  479  983  751  894
670  259  248  757  629  306  606  990
738  516  414  262  116  825  181  134
343   22  233  536  760  979   71  201
336   61  160    5  729  644  475  993
```

49. Rewrite the straight insertion sort from Chapter 8 using pointer arithmetic. The data to be sorted are to be read from a file. The array is to be

dynamically allocated in the heap after reading the file to determine the number of elements. While reading the data to determine the size of array you will require, print them 10 integers to a line. Use the test data shown below.

838	758	113	515	51	627	10	419	212	86
749	767	84	60	225	543	89	183	137	566
966	978	495	311	367	54	31	145	882	736
524	505	394	102	851	67	754	653	561	96
628	188	85	143	967	406	165	403	562	834
353	920	444	803	962	318	422	327	457	945
479	983	751	894	670	259	248	757	629	306
606	990	738	516	414	262	116	825	181	134
343	22	233	536	760	979	71	201	336	61

The data will be sorted as they are read into the array. *Do not fill the array and then sort the data.* After the array has been sorted, print the data again using the same format you used for the unsorted data.

50. Write a program to answer inquiries about student data. Using a menu-driven user interface, provide the capability to print out the scores, average, or grade for a student. A fourth menu option is to provide all information about a given student. All array functions are to receive the array as a pointer and use pointer arithmetic.

The data in Table 10-4 will be stored in a two-dimensional array.

Student	Quiz 1	Quiz 2	Quiz 3	Quiz 4	Quiz 5
1234	52	7	100	78	34
1947	45	40	88	78	55
2134	90	36	90	77	30
2877	55	50	99	78	80
3124	100	45	20	90	70
3189	22	70	100	78	77
4532	11	17	81	32	77
4602	89	50	91	78	60

continued

TABLE 10-4 Student Data for Project 50

Student	Quiz 1	Quiz 2	Quiz 3	Quiz 4	Quiz 5
5405	11	11	0	78	10
5678	20	12	45	78	34
6134	34	80	55	78	45
6999	0	98	89	78	20
7874	60	100	56	78	78
8026	70	10	66	78	56
9893	34	9	77	78	20

TABLE 10-4 Student Data for Project 50 *(continued)*

51. Contract bridge is a popular card game played by millions of people throughout the world. It began in the 1920s as a variation of an old English card game, whist. In bridge, the entire deck is dealt to four players named North, South, East, and West. In tournament bridge, teams of players (North-South versus East-West) compete with other players using the same hands (sets of 13 cards). Today it is common for large tournaments to use computer-generated hands. Write a program to shuffle and deal the hands for one game.

To simulate the bridge deck, use an array of 52 integers initialized from 1 to 52. To shuffle the deck, loop through the array, exchanging the current card element with a random element (use the random number generator discussed in Chapter 4). After the deck has been shuffled, print the hands in four columns using the player's position as a heading. Use the following interpretation for the cards' suits:

1 to 13	Clubs
14 to 26	Diamonds
27 to 39	Hearts
40 to 52	Spades

To determine the rank of the card, use its number (modulo 13) + 1. The interpretation of the rank follows: 1 is an ace, 2 through 10 have their value, 11 is a jack, 12 is a queen, and 13 is a king.

52. Modify Project 51 to sort each hand by suit (clubs lowest and spades highest) and within suit, by rank after the shuffle. Hint: After shuffling the deck, consider the first 13 cards the hand for North, the next 13 cards the hand for East, and so forth. Then use a function to sort each hand in turn.

Chapter 11

Strings

It is impossible to write a well-structured and human-engineered program without using strings. Although you probably weren't aware of it, you have been using strings ever since you wrote your first C program.

Some languages, such as Pascal, Ada, and C++ provide intrinsic string types, but C does not. In C, the programmer implements strings. Because strings are so important, however, functions to manipulate them have been defined in an ad hoc standard library.

In this chapter we first consider how strings are defined and stored, and then we explore the standard string functions that are available in C.

Objectives

- ❑ To understand design concepts for fixed-length and variable-length strings
- ❑ To understand the design implementation for C-language delimited strings
- ❑ To write programs that read, write, and manipulate strings
- ❑ To write programs that use the string functions in the string library
- ❑ To write programs that use arrays of strings
- ❑ To write programs that parse a string into separate variables
- ❑ To understand the software engineering concepts of information hiding and cohesion

11.1 String Concepts

In general, a string is a series of characters treated as a unit. Computer science has long recognized the importance of strings, but it has not adapted a standard for their implementation. We find, therefore, that a string created in Pascal differs from a string created in C.

Virtually all string implementations treat a string as a variable-length piece of data. Consider, for example, one of the most common of all strings, a name. Names, by their very nature, vary in length. It makes no difference if we are looking at the name of a person, a textbook, or an automobile.

Given that we have data that can vary in size, how do we accommodate them in our programs? We can store them in fixed-length objects, or we can store them in variable-length objects. This breakdown of strings is seen in Figure 11-1.

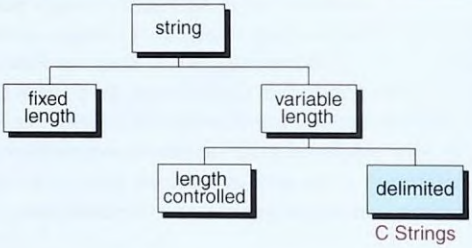


FIGURE 11-1 String Taxonomy

Fixed-Length Strings

When implementing a **fixed-length string** format, the first decision is the size of the variable. If we make it too small, we can't store all the data. If we make it too big, we waste memory.

Another problem associated with storing variable data in a fixed-length data structure is how to tell the data from the nondata. A common solution is to add nondata characters, such as spaces, at the end of the data. Of course, this means that the character selected to represent the nondata value cannot be used as data.

Variable-Length Strings

A much preferred solution is to create a structure that can expand and contract to accommodate the data. Thus, to store a person's name that consists of only one letter, we would provide only enough storage for one character.[1]

[1] The shortest name that we are aware of is O. To accommodate the computers of credit card and other companies, however, Mr. O was forced to legally change his name to Oh.

To store a person's name that consists of 30 characters, the structure would be expanded to provide storage for 30 characters.

This flexibility does not come without a cost, however. There must be some way to tell when we get to the end of the data. Two common techniques are to use length-controlled strings and delimited strings.

Length-Controlled Strings

Length-controlled strings add a count that specifies the number of characters in the string.This count is then used by the string manipulation functions to determine the actual length of the data.

Delimited Strings

Another technique used to identify the end of the string is the **delimiter** at the ends of **delimited strings**. You are already familiar with the concept of delimiters, although you probably don't recognize them as such. In English, each sentence, which is a variable-length string, ends with a delimiter, the period. Commas, semicolons, colons, and dashes are other common delimiters found in English.

The major disadvantage of the delimiter is that it eliminates one character from being used for data. The most common delimiter is the ASCII null character, which is the first character in the ASCII character sequence (`\0`). This is the technique used by C.

Figure 11-2 shows length-controlled and delimited strings in memory.

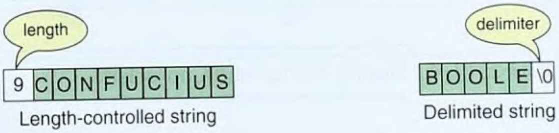


FIGURE 11-2 String Formats

11.2 C Strings

A C string is a variable-length array of characters that is delimited by the null character. Generally, string characters are selected only from the printable character set. Nothing in C, however, prevents any character, other than the null delimiter, from being used in a string. In fact, it is quite common to use formatting characters, such as tabs, in strings.

C uses variable-length, delimited strings.

Storing Strings

In C, a string is stored in an array of characters. It is terminated by the null character ('\0'). Figure 11-3 shows how a string is stored in memory. What precedes the string and what follows it is not important. What is important is that the string is stored in an array of characters that ends with a null delimiter. Because a string is stored in an array, the name of the string is a pointer to the beginning of the string.

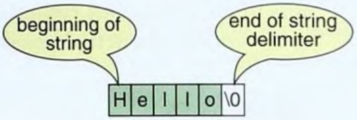


FIGURE 11-3 Storing Strings

Figure 11-4 shows the difference between a character stored in memory and a one-character string stored in memory. The character requires only one memory location. The one-character string requires two memory locations: one for the data and one for the delimiter. The figure also shows how an empty string is stored. Empty strings require only the end-of-string marker.

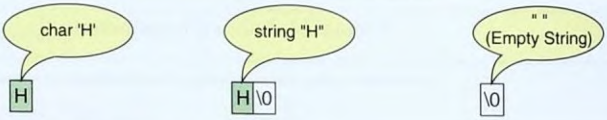


FIGURE 11-4 Storing Strings and Characters

The String Delimiter

At this point, you may be wondering, "Why do we need a null character at the end of a string?" The answer is that a string is not a data type but a data structure. This means that its implementation is logical, not physical. The physical structure is the array in which the string is stored. Since the string, by its definition, is a variable-length structure, we need to identify the logical end of the data within the physical structure.

Looking at it another way, if the data are not variable in length, then we don't need the string data structure to store them. They are easily stored in an array, and the end of the data is always the last element in the array. But, if the data length is variable, then we need some other way to determine the end of the data.

The null character is used as an end-of-string marker. It is the sentinel used by the standard string functions. In other words, the null character at the end lets us treat the string as a sequence of objects (characters) with a defined object at the end that can be used as a sentinel. Figure 11-5 shows the difference between an array of characters and a string.

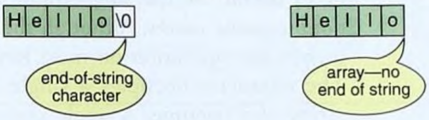


FIGURE 11-5 Differences Between Strings and Character Arrays

Because strings are variable-length structures, we must provide enough room for the maximum-length string we will have to store, plus one for the delimiter. It is possible that the structure will not be filled, so we can have an array with the null character in the middle. In this case, we treat the part of the array from the beginning to the null character as the string and ignore the rest. In other words, any part of an array of characters can be treated as a string as long as the string ends in a null character. This is shown in Figure 11-6.

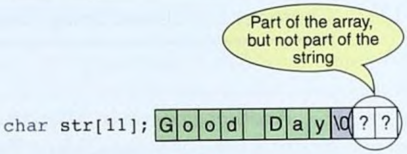


FIGURE 11-6 Strings in Arrays

String Literals

A string literal—or as it is also known, string constant—is a sequence of characters enclosed in double quotes. For example, each of the following is a string literal:

```
"C is a high-level language."
"Hello"
"abcd"
```

When string literals are used in a program, C automatically creates an array of characters, initializes it to a null-delimited string, and stores it, remembering its address. It does all this because we use the double quotes that immediately identify the data as a string value.

A string literal is enclosed in double quotes.

Strings and Characters

When all we need to store is a single character, we have two options: We can store the data as a character literal or as a string literal. To store it as a character literal, we use *single quote marks*. To store it as a string literal, we use *double quote marks*. Although the difference when we code the literal is only a shift-key operation on most keyboards, the difference in memory is great. The character occupies a single memory location. The data portion of the string also occupies a single memory location, but there is an extra memory location required for the delimiter.

The differences in the ways we manipulate the data are even greater. For example, moving a character from one location to another requires only an assignment. Moving a string requires a function call. It is important, therefore, that you clearly understand the differences. Figure 11-7 shows examples of both character literals and string literals.

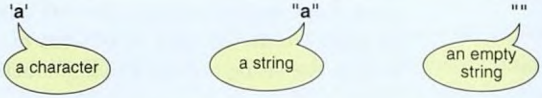


FIGURE 11-7 Character Literals and String Literals

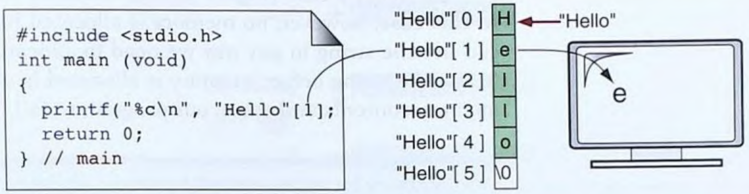
Another important difference between a string and a character is how we represent the absence of data. Technically, there is no such thing as an empty character. Logically, we often specify that a space (' ') or a null character ('\0') represents the absence of data. Since the character exists in all cases, however, both of these concepts require that we program for the interpretation of no data.

A string, on the other hand, can be empty. That is, since it is a variable-length structure, a string can exist with no data in it. A string that contains no data consists of only a delimiter. This concept is specified in the definition of a string and is programmed into all the string-handling functions. We can, therefore, move or compare an empty string without knowing that we are dealing with no data. An example of a null string is also shown in Figure 11-7.

Referencing String Literals

A string literal is stored in memory. Just like any object stored in memory, it has an address. Thus, we can refer to a string literal by using pointers.

Let's first examine addressing a string literal. The literal, since it is an array of characters, is itself a pointer constant to the first element of the string. Generally, when we use it, we are referring to the entire string. It is possible, however, to refer to only one of the characters in the string, as shown in Figure 11-8.



```
#include <stdio.h>
int main (void)
{
   printf("%c\n", "Hello"[1]);
   return 0;
} // main
```

FIGURE 11-8 String Literal References

Declaring Strings

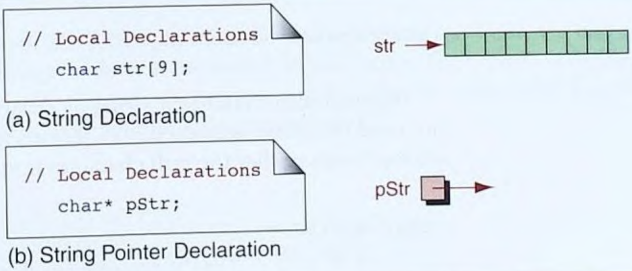
As we said, C has no string type. To declare a string, therefore, we must use one of the other available structures. Since strings are a sequence of characters, it is only natural that the structure used to store string variables is the character array.

In defining the array to store a string, we must provide enough room for the data and the delimiter. The storage structure, therefore, must be one byte larger than the maximum data size. A string declaration for an eight-character string, including its delimiter, is shown below.

```
char str[9];
```

As we have seen, string declaration defines memory for a string when it is declared as an array in local memory (the stack). However, we can also declare a string as a pointer. When we declare the pointer, however, memory is allocated only for the pointer; no memory is allocated for the string itself. In this case, we must allocate memory for the string either dynamically or using a string literal.

Figure 11-9 demonstrates two different ways to declare and define a string. Let's examine each carefully. In the first case (Figure 11-9a), memory is allocated for future characters. The name of the string is a pointer constant. We don't need to worry about allocating memory because the declaration is for an array and memory allocation is automatic. We can read data into the string and change data in the string as necessary.



```
// Local Declarations
   char str[9];
```

(a) String Declaration

```
// Local Declarations
   char* pStr;
```

(b) String Pointer Declaration

FIGURE 11-9 Defining Strings

The second case (Figure 11-9b) allocates memory for a pointer variable. In this case, however, no memory is allocated for the string itself. Before we can use the string in any way we need to allocate memory for it. Any attempt to use the string before memory is allocated is a logic error that may destroy memory contents and cause our program to fail.

> **Memory for strings must be allocated before the string can be used.**

Initializing Strings

We can initialize a string the same way that we initialize any storage structure by assigning a value to it when it is defined. In this case, the value is a string literal. For example, to assign "Good Day" to a string, we would code

```
char str[9] = "Good Day";
```

Since a string is stored in an array of characters, we do not need to indicate the size of the array if we initialize it when it is defined. For instance, we could define a string to store the month January, as shown below.

```
char month[] = "January";
```

In this case, the compiler will create an array of 8 bytes and initialize it with January and a null character. We must be careful, however, because `month` is an array. If we now tried to store "December" in it, we would overrun the array and destroy whatever came after the array. This example points out one of the dangers of strings: We must make them large enough to hold the longest value we will place in the variable.

C provides two more ways to initialize strings. A common method is to assign a string literal to a character pointer, as shown below. This creates a string for the literal and then stores its address in the string pointer variable, `pStr`. To clearly see the structure, refer to Figure 11-10.

```
char* pStr = "Good Day";
```

We can also initialize a string as an array of characters. This method is not used too often because it is so tedious to code. Note that in this example, we must ensure that the null character is at the end of the string.

```
char str[9] =
    {'G','o','o','d',' ','D','a','y', '\0'};
```

The structures created by these three examples are shown in Figure 11-10.

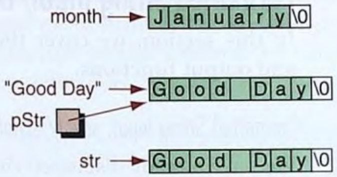


FIGURE 11-10 Initializing Strings

Strings and the Assignment Operator

Since the string is an array, the name of the string is a pointer constant. As a pointer constant, it is an rvalue and therefore cannot be used as the left operand of the assignment operator. This is one of the most common errors in writing a C program; fortunately, it is a compile error, so it cannot affect our program.

```
char str1[6] = "Hello";
char str2[6];
str2 = str1;                                    // Compile error
```

Although we could write a loop to assign characters individually, there is a better way. C provides a rich library of functions to manipulate strings, including moving one string to another. We discuss this library in the section "String Manipulation Functions."

Reading and Writing Strings

A string can be read and written. C provides several string functions for input and output. We discuss them in the following section, "String Input/Output Functions."

11.3 String Input/Output Functions

C provides two basic ways to read and write strings. First, we can read and write strings with the formatted input/output functions, *scanf/fscanf* and *printf/fprintf*. Second, we can use a special set of string-only functions, get string (*gets/fgets*) and put string (*puts/fputs*).[2]

[2] C provides a parallel set of read functions for wide characters. They function the same except for the type.

Formatted String Input/Output

In this section, we cover the string-related portions of the formatted input and output functions.

Formatted String Input: *scanf/fscanf*

We have already discussed the basic operations of the format input functions (see Chapter 7). However, two conversion codes pertain uniquely to strings, and we discuss them here.

The String Conversion Specification

We read strings using the read-formatted function (*scanf*). The conversion code for a string is "s." The *scanf* functions then do all the work for us. First, they skip any leading whitespace. Once they find a character, they read until they find whitespace, putting each character in the array in order. When they find a trailing whitespace character, they end the string with a null character. The whitespace character is left in the input stream. To delete the whitespace from the input stream, we use a space in the format string before the next conversion code or FLUSH the input stream, whichever is more appropriate.

The conversion specification strings can use only three options fields, flag, maximum field size, and size.

The string conversion code(s) skips whitespace.

Flag

Like all inputs, the only flag that can be used is the asterisk (*), which discards the data read.

Maximum Field Size

When present, the maximum field size specifies the maximum number of characters that can be read.

Size

If no size option is specified, we read normal characters. To read wide characters, we use size l (ell).

Examples

For example, to read a string, such as month, from the keyboard, we could simply write the statement shown below.

```
scanf("%s", month);
```

An address operator is not required for month, since it is already a pointer constant. In fact, it would be an error to use one. The only thing we need to worry about is to make sure that the array is large enough to store all the data. If it isn't, then we destroy whatever follows the array in memory. Therefore, we must make sure we don't exceed the length of the data. Assuming that month has been defined as

```
char month[10];
```

we can protect against the user entering too much data by using a width in the field specification. (Recall that the width specifies the *maximum* number of characters to be read.) The modified *scanf* statement is shown below.

```
scanf("%9s", month);
```

Note that we set the maximum number of characters at nine while the array size is ten. This is because *scanf* will read up to nine characters and then insert the null character. Now, if the user accidentally enters more than nine characters, the extra characters will be left in the input stream. But this can cause a problem. Assuming that the data are being entered as a separate line—that is, that there is only one piece of data on the line—we use the preprocessor-defined statement, FLUSH—see Chapter 7, to eliminate any extra characters that were entered. This function also flushes the newline that is left in the input stream by *scanf* when the user correctly enters data. The complete block of code to read a month is shown in Program 11-1.

PROGRAM 11-1 Reading Strings

```
{ // Read Month
  #define FLUSH while (getchar() != '\n')
  char month[10];

  printf("Please enter a month. ");
  scanf("%9s", month);
  FLUSH;
} // Read Month
```

The Scan Set Conversion Code ([...])

In addition to the string conversion code, we can also use a scan set conversion code to read a string. The scan set conversion specification consists of the open bracket ([), followed by the edit characters, and terminated by the close bracket (]). The characters in the scan set identify the valid characters, known as the scan set, that are to be allowed in the string. All characters except the close bracket can be included in the set.

Edited conversion reads the input stream as a string. Each character read by *scanf*/*fscanf* is compared against the scan set. If the character just read is in the scan set, it is placed in the string and the scan continues. The first character that does not match the scan set stops the read. The nonmatching character remains in the input stream for the next read operation. If the first character read is not in the scan set, the *scanf/fscanf* terminates and a null string is returned.

A major difference between the scan set and the string conversion codes is that the scan set does not skip leading whitespace. Leading whitespace is either put into the string being read when the scan set contains the corresponding whitespace character, or stops the conversion when it is not.

The edit set does not skip whitespace.

In addition to reading a character that is not in the scan set, there are two other terminating conditions. First, the read will stop if an end-of-file is detected. Second, the read will stop if a field width specification is included and the maximum number of characters has been read.

For example, let's assume we have an application that requires we read a string containing only digits, commas, periods, the minus sign, and a dollar sign; in other words, we want to read a dollar value as a string. No other characters are allowed. Let's also assume that the maximum number of characters in the resulting string is 10. The format string for this operation would be[3]

```
scanf("%10[0123456789.,-$]", str);
```

Sometimes it is easier to specify what is not to be included in the scan set rather than what is valid. For instance, suppose that we want to read a whole line. We can do this by stating that all characters except the newline (\n) are valid. To specify invalid characters, we start the scan set with the caret (^) symbol. The caret is the negation symbol and in effect says that the following characters are not allowed in the string. (If you know UNIX, this should sound familiar.) To read a line, we would code the *scanf* as shown below.

```
scanf("%81[^\n]", line);
```

In this example, *scanf* reads until it finds the newline and then stops. Note that we have again set the width of the data to prevent our string, `line`, from being overrun. We would never use this code, however. As we see in the next section, an intrinsic string function does it for us.

For the last example, let's read a 15-character string that can have any character except the special characters on the top of the keyboard. In this case, we again specify what is not valid. This conversion code is shown in the following example.

```
scanf("%15[^~!@#$%^&*()_+]", str);
```

[3] UNIX users note that the dash (–) does not have the same meaning in the scan set that it does in UNIX.

Note that the caret can be included in the scan set, as long as it is not the first character. Similarly, the closing bracket is treated as text as long as it is the first character after the opening bracket. However, if it follows the negation symbol (^), then it ends the scan set. This means that the close bracket cannot be used when the scan set is being negated. The following example reads a string containing brackets and digits only.

```
scanf("%15[][0123456789]", str);
```

Always use a width in the field specification when reading strings.

Formatted String Output: *printf/fprintf*

Formatted string output is provided in the *printf* and *fprintf* functions. They use the same string conversion codes that we used for string input.

C has four options of interest when we write strings using these print functions: the left-justify flag, width, precision, and size. The left-justify flag (–) and the width are almost always used together.

Justification Flag

The justification flag (–) is used to left justify the output. It has meaning only when a width is also specified, and then only if the length of the string is less than the format width. Using the justification flag results in the output being left justified, as shown below. If no flag is used, the justification is right.

```
printf("|%-30s|\n", "This is the string");
Output:
|This is the string            |
```

Minimum Width

The width sets the *minimum* size of the string in the output. If it is used without a flag, the string is printed right justified as shown below.

```
printf("|%30s|\n", "This is the string");
Output:
|            This is the string|
```

Precision

Strings can get quite long. Because the width is the minimum number of characters that can be printed, a long string can easily exceed the width specified, which destroys column formatting. Therefore, C also uses the precision option to set the maximum number of characters that will be written. In the following typical example, we set the maximum characters to one less than the width to ensure a space between the string and the next column.

```
printf("|%-15.14s|", "12345678901234567890");
Output:
|12345678901234 |
```

The maximum number of characters to be printed is specified by the precision in the format string of the field specification.

Size

If no size option is specified, we write normal characters. To write wide characters, we use size `l` (ell).

String Input and Output Examples

EXAMPLE 11-1

Read Part of Line

When data are stored in secondary files, the data we need are often only a part of a file. Program 11-2 demonstrates how to use the scan set to read part of a line and discard the rest. In the following example, the input contains numeric data, with each line containing two integers and a float. We are interested only in the second integer; the rest of each line will be discarded. As we wrote the program, it could contain any type or amount of data after the second integer. It is important to note, however, that if there are not two integers at the beginning of the line, the program fails.

PROGRAM 11-2 Demonstrate String Scan Set

```
/* Read only second integer.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
// Local Declarations
   int   amount;
   FILE* spData;

// Statements
   if (!(spData = fopen ("P11-03.TXT", "r")))
```

continued

PROGRAM 11-2 Demonstrate String Scan Set *(continued)*

```
16          {
17           printf("\aCould not open input file.\n");
18           exit (100);
19          } // if
20       // Read and print only second integer
21       while (fscanf(spData,
22              " %*d%d%*[^\n]", &amount) != EOF)
23           printf("Second integer: %4d\n", amount);
24
25       printf("End of program\n");
26       fclose (spData);
27       return 0;
28    }  // main
```

```
Input:
123 456 7.89
987 654 3.21
Results:
Second integer: 456
Second integer: 654
End of program
```

Program 11-2 Analysis As long as two numbers are at the beginning of each line, this program works just fine. It skips the first number, reads the number into `amount`, and then discards the rest of the line. Study the flags contained in the format string in statement 22 carefully. It consists of only three field specifications. The first one reads and discards an integer. The second one reads `amount`. The third one considers the rest of the line as a long string; *fscanf* reads until it finds a newline. Because we are not interested in these data for the first and third parts of the line, we used the suppress flag to discard them.

EXAMPLE 11-2 Delete Leading Whitespace from Line

A similar problem occurs when a file contains leading whitespace at the beginning of a line. In Program 11-3, we first determine if there is a whitespace character at the beginning of the line. If there is, we delete all leading spaces using the scan set and then read the rest of the line. Its only limitation is that the nonspace portion of the line cannot be longer than 80 characters.

PROGRAM 11-3 Delete Leading Whitespace

```
/* Delete leading spaces at beginning of line.
      Written by:
      Date:
*/
```

continued

PROGRAM 11-3 Delete Leading Whitespace *(continued)*

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

int main (void)
{
// Local Declarations
   char line[80];

// Statements
   printf("Enter data:  ");
   while ((fscanf(stdin, "%*[\t\v\f ]%79[^\n]", line))
                         != EOF)
      {
       printf("You entered: %s\n", line);

       // Discard newline and set line to null string
       fgetc (stdin);
       *(line) = '\0';
       printf("Enter data:  ");
      } // while

    printf("\nThank you\n");
    return 0;
}  // main
```

```
Results:
Enter data:  No whitespace here.
You entered: No whitespace here.
Enter data:   Only one whitespace character.
You entered: Only one whitespace character.
Enter data:        Tabs and spaces here.
You entered: Tabs and spaces here.
Enter data:  Next line is only one space.
You entered: Next line is only one space.
Enter data:
You entered:
Enter data:  ^d
Thank you
```

EXAMPLE 11-3 Read Student Names and Scores

Very often reading names requires that the names be parsed into first and last names. Parsing names presents unique problems. First, we must skip the whitespace between the parts of the name and the newline at the end of the line. The string format code (s) stops when it reads a whitespace character. Second, some names have special characters such as a dash in them.

In Program 11-4 we read a file of student names and scores. As each name is read, it is printed. We read the first and last names using the string conversion code. We read the student's score as an integer. As an alternative, we could have used the scan-set conversion code, but it is simpler to use the basic string code (remember KISS).

PROGRAM 11-4 Read Student Names and Scores

```
/* Demonstrate reading names from a file.
   Written by:
   Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main (void)
{
// Local Declarations
   char  first[80];
   char  last[80];
   int   score;
   FILE* spStuScores;

// Statements
   if (!(spStuScores = fopen ("P11-04.TXT", "r")))
      {
       printf("\aCould not open student file.\n");
       exit (100);
      } // if

   // Read and print first name, last name, and score
   while (fscanf(spStuScores, " %s %s %d",
           first, last, &score) == 3)
      printf("%s %s %3d\n", first, last, score);

   printf("End of Student List\n");
   fclose (spStuScores);
   return 0;
}  // main
```

```
Results:
George Washington  95
Benedict Arnold  53
Mary Todd-Lincoln  91
End of Student List
```

Program 11-4 Analysis There is only one significant point to note in this program—how we handle end of data. As long as we read three pieces of data, we continue looping. When the last data have been read, the scan returns 0 and the *while* statement becomes false.

String Input/Output

In addition to the formatted string functions, C has two sets of string functions that read and write strings without reformatting any data. These functions convert text-file lines to strings and strings to text-file lines. A line consists of a string of characters terminated by a newline character.

C provides two parallel sets of functions, one for characters and one for wide characters. They are virtually identical except for the type. Because wide characters are not commonly used and because the functions operate identically, we limit our discussion to the character type. The wide-character functions are listed in Appendix F.

Line to String

The *gets* and *fgets* functions take a line (terminated by a newline) from the input stream and make a null-terminated string out of it. They are therefore sometimes called line-to-string input functions.

The function declarations for get string are shown below.

```
char*    gets   (char*    strPtr);
char*    fgets  (char*    strPtr, int size, FILE* sp);
```

Figure 11-11 shows the concept. As you can see, *gets* and *fgets* do not work the same. The *gets* function converts the return (newline character) to the end-of-string character (\0), while *fgets* puts it in the string and appends an end-of-string delimiter.

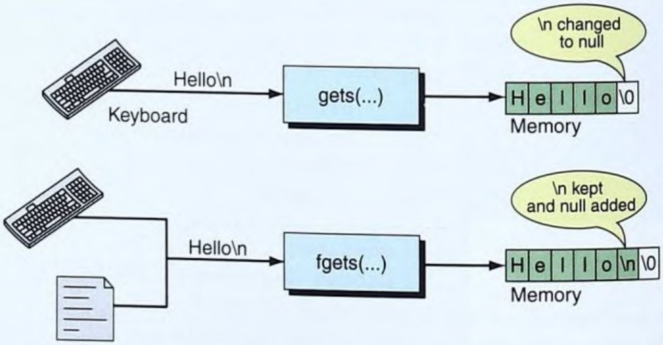


FIGURE 11-11 *gets* and *fgets* Functions

The source of data for the *gets* is standard input; the source of data for *fgets* can be a file or standard input. Both accept a string pointer and return the same pointer if the input is successful. If any input problems occur, such as detecting end-of-file before reading any data, they return *NULL*. If no data were read, the input area is unchanged. If an error occurs after some data have been read, the contents of the read-in area cannot be determined. The current string may or may not have a valid null character.

Note that since no size is specified in *gets,* it reads data until it finds a newline or until the end-of-file. If a newline character is read, it is discarded and replaced with a null character.

The *fgets* function requires two additional parameters: one specifying the array size that is available to receive the data, and the other a stream. It can be used with the keyboard by specifying the *stdin*. In addition to newline and end of file, the reading will stop when `size - 1` characters have been read.

Since no length checking is possible with *gets*, we recommend that you never use it. Should a user enter too much data, you will destroy the data after the string input area, and your program will not run correctly. Rather, use *fgets*, and specify the standard input file pointer (*stdin*).

EXAMPLE 11-4 Demonstrate *fgets* Operation

Now let us write a simple program that uses *fgets*. In Program 11-5, we use *fgets* to read a string and then print it.

PROGRAM 11-5 Demonstrate *fgets* Operation

```
/* Demonstrate the use of fgets in a program
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   char str[81];

// Statements
   printf("Please enter a string: ");
   fgets (str, sizeof (str), stdin);
   printf("Here is your string: \n\t%s", str);
   return 0;
}  // main
```

```
Results:
Please enter a string: Now is the time for all students
```

continued

PROGRAM 11-5 Demonstrate *fgets* Operation *(continued)*

```
Here is your string:
     Now is the time for all students
```

String to Line

The *puts/fputs* functions take a null-terminated string from memory and write it to a file or the keyboard. Thus, they are sometimes called string-to-line output functions.

Figure 11-12 shows how these functions work. All change the string to a line. The null character is replaced with a newline in *puts*; it is dropped in *fputs*. Because *puts* is writing to the standard output unit, usually a display, this is entirely logical. On the other hand, *fputs* is assumed to be writing to a file where newlines are not necessarily required. It is the programmer's responsibility to make sure the newline is present at the appropriate place.

Note how the newline is handled in these functions. Then compare their use of the newline to the *gets* and *fgets* functions. While the output functions treat the newline the opposite of the input functions, they are compatible. As long as we are reading from a file and writing to a file, the newlines will be handled automatically. Care must be taken, however, when we read from the keyboard and write to a file.

The declarations for these functions are shown below.

```
int puts  (const char* strPtr);
int fputs (const char* strPtr, FILE* sp);
```

The string pointed to by `strPtr` is written to the indicated file as explained above. If the write is successful, it returns a non-negative integer; if any transmission errors occur, it returns `EOF`. Note that the absence of a null character to terminate the string is not an error; however, it will most likely cause your program to fail.

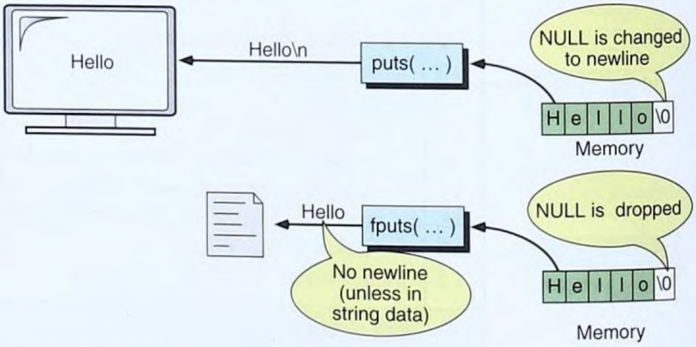


FIGURE 11-12 *puts* and *fputs* Operations

Examples

This section contains examples to demonstrate the use of these string functions.

EXAMPLE 11-5 The following program demonstrates the use of *fputs* with *stdout*. It calls *fputs* three times. The first time, we pass a pointer to the beginning of an array. The third time, we pass a pointer that is at the middle of the array. If you were to run this code, you would see that it does not matter where a string pointer starts. The function starts at the address in the pointer and writes until it finds a null character. Note that we had to include a newline put because *fputs* does not insert a newline. The output is shown in Program 11-6.[4]

PROGRAM 11-6 Demonstration of Put String

```
/* Demonstrate fput string
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Definitions
   char str[] = "Necessity Is the Mother of Invention.";
   char* pStr = str;

// Statements
   fputs(pStr, stdout);
   fputs("\n", stdout);
   fputs(pStr + 13, stdout);
   return 0;
}  // main
```

```
Results:
Necessity Is the Mother of Invention
the Mother of Invention.
```

EXAMPLE 11-6 Typewriter Program

Program 11-7 plays the role of a line-at-a-time typewriter. In other words, it accepts text, line by line, from the keyboard and writes it to a text file. The program stops when it detects an end-of-file.

[4] The quote in Program 11-6 is attributed to the American economist Thorstein Veblen.

PROGRAM 11-7 Typewriter Program

```
 1  /* This program creates a text file from the keyboard.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6  #include <stdlib.h>
 7
 8  int main (void)
 9  {
10  // Local Declarations
11     char  str[100];
12     FILE* spOut;
13
14  // Statements
15     if (!(spOut = fopen ("P11-07.TXT", "w")))
16        {
17         printf("\aCould not open output file.\n");
18         exit (100);
19        } // if
20     while (fgets(str, sizeof (str), stdin))
21        fputs(str,  spOut);
22     fclose (spOut);
23     return 0;
24  }  // main
```

Program 11-7 Analysis Note that we are reading the keyboard (*stdin*) using *fgets* (see statement 20). This requires that we specify the file name, even though we do not have to declare it or open it. By using *fgets* we are able to ensure that the user will not overrun our string variable area (`str`). But even if the user enters too much data, nothing is lost. The data are left in the input stream buffer and are read in the next loop iteration.

This little program has two problems, one stylistic and one technical. The style problem is that it contains no instructions to the user, who must therefore guess what is to be done. This leads directly to the second problem. Some systems have a problem if the user should end the program with an end-of-file but no return. Depending on the system, the last line is lost or the system may just wait for an end-of-file at the beginning of the line. This problem can be prevented by adding good user instructions and a prompt.

EXAMPLE 11-7 Print Selected Lines

Program 11-8 reads text from the keyboard, line by line, and prints only the lines that start with uppercase letters. In this case we will write to the standard output (*stdout*) file. This will allow us to direct the output to the printer by assigning standard output to a printer. If standard output is assigned to the monitor, then the input and output will be interleaved, as shown in the results. The output lines are green.

PROGRAM 11-8 Print Selected Sentences

```
/* Echo keyboard input that begins with capital letter.
      Written by:
      Date written:
*/
#include <ctype.h>
#include <stdio.h>

int main (void)
{
// Local Declarations
   char strng[81];

// Statements
   while (fgets (strng, sizeof(strng), stdin))
       if (isupper(*strng))
           fputs(strng, stdout);
   return 0;
}  // main
```

```
Results:
Now is the time
Now is the time
for all good students
to come to the aid
of their school.
Amen
Amen
```

Program 11-8 Analysis In this program, we use the character function, *isupper*, to determine which lines we want to write. The output lines are printed in color. Although we are guarding against excessive input, we do not flush the line. If the user enters very long lines, they will be obvious when the program runs.

If you use this program or any variation of it to write to your printer, you will need to assign the printer to the standard output file. Refer to the documentation for your system to determine how to do this.

EXAMPLE 11-8 Print File Double Spaced

Program 11-9 reads a single-spaced text from a file and prints the text double spaced. In other words, it inserts a blank line after each line. In this program we direct the output to the standard output file, *stdout*, which is usually the monitor. To get the output to a printer, you need to redirect the output or assign *stdout* to the printer.

PROGRAM 11-9 Print File Double spaced

```
/* Write file double spaced.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
// Local Declarations
   char  strng[81];
   FILE* textIn;

// Statements
   if (!(textIn = fopen("P11-07.TXT", "r")))
      {
       printf("\aCan't open textdata\n");
       exit (100);
      } // if
   while (fgets(strng, sizeof(strng), textIn))
      {
       fputs(strng, stdout);
       putchar ('\n');
      } // while
   return 0;
}  // main
```

Program 11-9 Analysis Because we are reading data from a text file, we use the *fgets* function. This function guarantees that we will not overrun our input string variable. Note how we used the *sizeof* operator to set the maximum number of characters to be read. Since *fgets* adds a newline character at the end of the data being read, we don't need to worry about adding one when we write. After writing the data string, we use a *putchar* to write the blank line.

11.4 Arrays of Strings

When we discussed arrays of pointers in Chapter 10, we introduced the concept of a ragged array. Ragged arrays are very common with strings. Consider, for example, the need to store the days of the week in their textual format. We could create a two-dimensional array of seven days by ten characters (Wednesday requires nine characters), but this wastes space.

It is much easier and more efficient to create a ragged array using an array of string pointers. Each pointer points to a day of the week. In this way each string is independent, but at the same time, they are grouped together through the array. In other words, although each string is independent, we can pass them as a group to a function by passing only the name of the array of pointers. Program 11-10 demonstrates this structure.

PROGRAM 11-10 Print Days of the Week

```
 1  /* Demonstrates an array of pointers to strings.
 2      Written by:
 3      Date written:
 4  */
 5  #include <stdio.h>
 6
 7  int main (void)
 8  {
 9  // Local Declarations
10     char*  pDays[7];
11     char** pLast;
12
13  // Statements
14     pDays[0] = "Sunday";
15     pDays[1] = "Monday";
16     pDays[2] = "Tuesday";
17     pDays[3] = "Wednesday";
18     pDays[4] = "Thursday";
19     pDays[5] = "Friday";
20     pDays[6] = "Saturday";
21
22     printf("The days of the week\n");
23     pLast = pDays + 6;
24     for (char** pWalker = pDays;
25                 pWalker <= pLast;
26                 pWalker++)
27          printf("%s\n", *pWalker);
28     return 0;
29  } // main
```

Program 11-10 Analysis To print the days of the week, we use a *for* loop. Since `pDays` is a pointer constant, we also need a pointer variable to use pointer arithmetic. The pointer variable must point to the strings through `pDays`, which means that it will be a pointer to a pointer, as seen in statement 11.

A point of efficiency is the way we handled the limit test in the *for* loop. We could have simply coded it as

```
pWalker <= pDays + 6
```

This would require, however, that the ending address be recalculated for each limit test. (A good optimizing compiler should recognize that `pDays` is a pointer constant and that the calculation needs to be done only once, but we can't be sure that such efficient code would in fact be generated.) Therefore, we calculate the ending address just once, before the *while* loop, and then we can be sure that the limit test will be efficient.

Study this code carefully. Note first that `pWalker` is a pointer to a pointer to a character. Then notice how it is used in the *for* statement. It is initialized to the first element in `pDays`, then it is incremented until it is no longer less than or equal to `pLast`. Finally, note how it is used in the *printf* statement. The *printf* syntax requires that the variable list contain the address of the string to be printed. But `pWalker` is a pointer to an address that in turn points to the string (a pointer to a pointer). Therefore, when we dereference `pWalker`, we get the pointer to the string, which is what *printf* requires. This example is diagrammed in Figure 11-13.

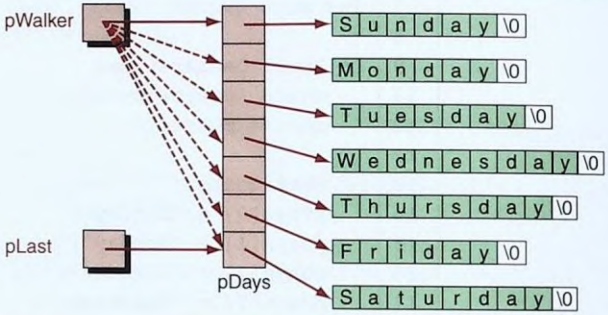


FIGURE 11-13 Pointers to Strings

11.5 String Manipulation Functions

Because a wstring is not a standard type, we cannot use it directly with most C operators. For example, to move one string to another, we must move the individual elements of the sending string to the receiving string. We cannot simply assign one string to another. If w4e were to write the move, we would have to put it in a loop.

C has provided a rich set of string functions. Besides making it easier for us to write programs, putting the string operations in functions provides the opportunity to make them more efficient when the operation is supported by hardware instructions. For example, computers often have a machine instruction that moves characters until a token, such as a null character, is reached. When this instruction is available, it allows a string to be moved in one instruction rather than in a loop.

In addition to the string character functions, C provides a parallel set of functions for wide characters. They are virtually identical except for the type. Because wide characters are not commonly used and because the functions operate identically, we limit our discussion to the character type. The wide-character functions are listed in Appendix F. The traditional string functions have a prefix of `str`. The basic format is shown in the following.

```
int str... (parameters)
```

The string character functions are found in the string library (*string.h*).

String Length

The string length function, *strlen*, returns the length of a string, specified as the number of characters in the string excluding the null character. If the string is empty, it returns zero. The function declaration is shown below.

```
int strlen (const char* string);
```

EXAMPLE 11-9 String Length Demonstration

To demonstrate the use of string length, let's write a program that reads text from the keyboard, line by line, and adds two blanks (spaces) at the beginning of each line before writing it to a file. In other words, it shifts each line two characters to the right. The code is shown in Program 11-11.

PROGRAM 11-11 Add Left Margin

```
/* Typewriter program: adds two spaces to the left
   margin and writes line to file
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main (void)
{
// Local Declarations
   FILE* spOutFile;
   char  strng[81];

// Statements
   if (!(spOutFile = fopen("P11-11.TXT", "w")))
      {
       printf("\aCould not open output file.\n");
       exit (100);
      } // if

   while (fgets(strng, sizeof(strng), stdin))
      {
```

continued

PROGRAM 11-11 Add Left Margin *(continued)*

```
25          fputc(' ', spOutFile);
26          fputc(' ', spOutFile);
27          fputs(strng, spOutFile);
28          if (strng[strlen(strng) - 1] != '\n')
29              fputs("\n", spOutFile);
30         } // while
31      fclose (spOutFile);
32      return 0;
33  }  // main
```

Program 11-11 Analysis To ensure that the user doesn't overrun the input area, we use the *fgets* function to read the keyboard. Some data may be left in the buffer, but we don't flush the buffer, because we want to write them to the file on the next line.

Also, since *fputs* does not add a newline when it writes the file, we need to ensure that there will be a new line at the end of each line. Therefore, in statement 28 we test the last character of the input string and if it isn't a newline, we write one.

Because we want to add characters to the beginning of the line, we must use a character operation. The function *fputc* writes one character to a designated file. To insert two characters, therefore, we use it twice, and then use *fputs* to write the line read by *fgets*.

String Copy

C has two string copy functions. The first, *strcpy,* copies the contents of one string to another. The second, *strncpy*, also copies the contents of one string to another, but it sets a maximum number of characters that can be moved. Therefore, *strncpy*, is a safer function.

Basic String Copy

The string copy function, *strcpy*, copies the contents of the from string, including the null character, to the string. Its function declaration is shown below.

```
char* strcpy (char* toStr, const char* fromStr);
```

If `fromStr` is longer than `toStr`, the data in memory after `toStr` are destroyed. It is our responsibility to ensure that the destination string array is large enough to hold the sending string. This should not be a problem, since we control the definition of both string variables. The address of `toStr` is returned, which allows string functions to be used as arguments inside other string functions. We will demonstrate the use of these returned pointers later in the chapter.

Figure 11-14 shows two examples of string copy. In the first example, the source string is shorter than the destination variable. The result is that, after the string has been copied, the contents of the last three bytes of `s1` are unchanged; `s1` is a valid string, however.

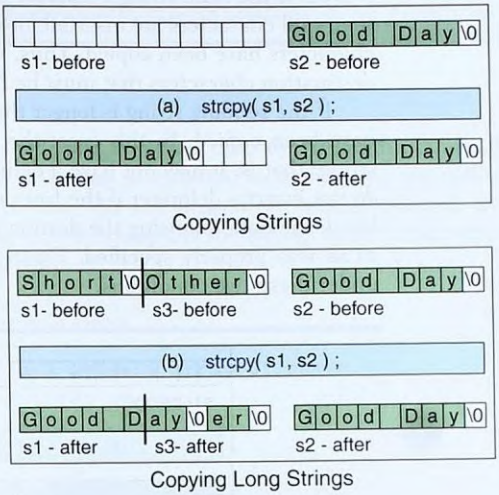


FIGURE 11-14 String Copy

In the second example in Figure 11-14, the destination variable, `s1`, is only 6 bytes, which is too small to hold all the data being copied from `s2` (8 bytes plus the delimiter). Furthermore, `s1` is immediately followed by another string, `s3`. Even though `s1` is too small, the entire string from `s2` is copied, partially destroying the data in `s3`. Although `s1` is a valid string, any attempt to access the data in `s3` will result in a string containing only `ay`.

If the source and destination strings overlap—that is, if they share common memory locations, then the results of the copy are unpredictable. In Figure 11-14, this would occur if we tried to execute the following statement:

```
strcpy((s2 + 2), s2);              // Invalid copy—overlap
```

String Copy—Length Controlled

Many of the problems associated with unequal string-array sizes can be controlled with the string-number copy function, *strncpy*. This function contains a parameter that specifies the *maximum* number of characters that can be moved at a time, as shown in its function declaration below.

```
char* strncpy (char* toStr, const char* fromStr,
               int   size);
```

In this function, `size` specifies the maximum number of characters that can be moved. Actually, the operation is a little more complex. If the size of the from string is equal to or greater than `size`, then `size` characters are moved. If the from string is smaller than size, the entire string is copied and then null characters are inserted into the destination string until exactly size characters have been copied. Thus, it is more correct to think of size as the destination characters that must be filled.

If the sending string is longer than size, the copy stops after size bytes have been copied. In this case, the destination variable may not be a valid string; that is, it may not have a delimiter. The string number copy functions *do not* insert a delimiter if the from string is longer than size. On the other hand, the data following the destination variable will be intact, assuming the size was properly specified. Figure 11-15 shows the operation of *strncpy* under these two conditions.

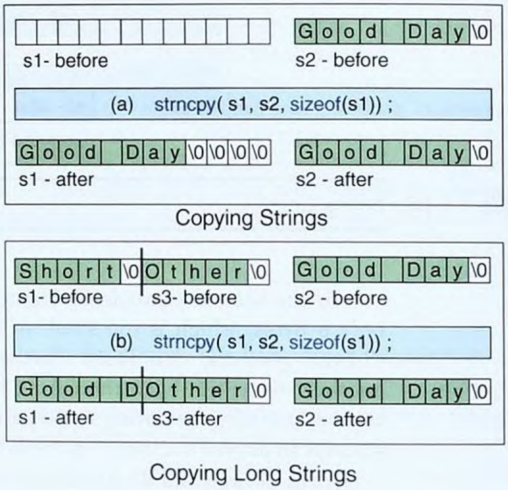


FIGURE 11-15 String-number Copy

We recommend that you always use *strncpy*; do not use *strcpy*. To prevent invalid strings, we also recommend that you move one fewer character than the maximum and then automatically place a null character in the last position. The code for this technique is shown below.

```
strncpy(s1, s2, sizeof(s1) -1);
*(s1 + (sizeof (s1) - 1)) = '\0';
```

Since the *strncpy* places null characters in all unfilled characters, we are guaranteed that the last character in the string array is a null character. If it is not, then the copy was short. By executing the above statements, we are assured that s1 will be a valid string, even if it doesn't have the desired contents.

A closing note: If size is zero or negative, nothing is copied. The destination string is unchanged.

Always use *strncpy* to copy one string to another.

EXAMPLE 11-10 Build Name Array

Let's write a small program that uses the *strcpy* function. Program 11-12 builds an array of strings in dynamic memory using the *calloc* function (see Chapter 10). It then fills the array from strings entered at the keyboard. When the array is full, it displays the strings to show that they were entered and stored correctly.

PROGRAM 11-12 Build Name Array in Heap

```
/* Build a dynamic array of names.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define FLUSH while (getchar() != '\n')

int main (void)
{
// Local Declarations
   char   input[81];
   char** pNames;            // array of pointers to char

   int size;
   int namesIndex;

// Statements
   printf("How many names do you plan to input? ");
   scanf ("%d", &size);
   FLUSH;

   // Allocate array in heap.
   // One extra element added for loop control
   pNames = calloc (size + 1, sizeof (char*));
   printf("Enter names:\n");

   namesIndex = 0;
   while (namesIndex < size
       && fgets(input, sizeof(input), stdin))
       {
```

continued

PROGRAM 11-12 Build Name Array in Heap *(continued)*

```
        *(pNames + namesIndex)  = (char*)
             calloc (strlen(input) + 1, sizeof(char));
        strcpy (*(pNames + namesIndex), input);
        namesIndex++;
      } // while

   *(pNames + namesIndex) = NULL;
   printf("\nYour names are: \n");
   namesIndex = 0;
   while (*(pNames + namesIndex))
      {
       printf("%3d: %s",
               namesIndex, *(pNames + namesIndex));
       namesIndex++;
      } // while
   return 0;
}  // main
```

```
Results:
How many names do you plan to input? 3
Enter names:
Tom
Rico
Huang

Your names are:
  0: Tom
  1: Rico
  2: Huang
```

Program 11-12 Analysis This little 50-line program contains some rather difficult code. To understand it better, let's look at the array structure, shown in Figure 11-16.

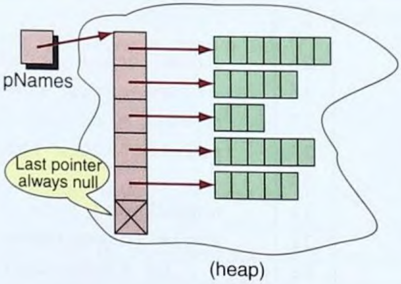


FIGURE 11-16 Structure for Names Array

The identifier, pNames, is a pointer to an array of pointers to a character that is dynamically allocated from the heap. Then, as each name is read, space is allocated from the heap, and its pointer is placed in the next location in the pNames array. The only way to refer to the names is by dereferencing pNames. To access an individual element, we use pNames and index it to get to an individual string pointer in the array. This code is shown in statement 36. Since the first parameter in the string copy is a pointer to a string, only one dereference is required.

To build the pointer array, we use a *while* loop with two limit tests: end-of-file and a full array. Either condition will stop the loading of the array. To print the array, however, we only need to test for a null pointer, since the pointer array is allocated with one extra element. Using an extra element is a common programming technique that makes processing arrays of pointers easier and more efficient.

String Compare

As with the string copy functions, C has two string compare functions.[5] The first, *strcmp*, compares two strings until unequal characters are found or until the end of the strings is reached. The second, *strncmp*, compares until unequal characters are found, a specified number of characters have been tested, or until the end of a string is reached.

Both functions return an integer to indicate the results of the compare. Unfortunately, the results returned do not map well to the true–false logical values that we see in the *if...else* statement (see Chapter 5), so you will need to memorize a new set of rules:

1. If the two strings are equal, the return value is *zero*. Two strings are considered equal if they are the same length and all characters in the same relative positions are equal.
2. If the first parameter is less than the second parameter, the return value is *less than zero*. A string, s1, is less than another string, s2, if starting from the first character, we can find a character in s1 that is less than the character in s2. Note that when the end of either string is reached, the null character is compared with the corresponding character in the other string.
3. If the first parameter is greater than the second parameter, the return value is *greater than zero*. A string, s1, is greater than string, s2, if starting from the first character, we can find a character in s2 that is greater than the corresponding character in s1. Note that when the end of either string is reached, the null character is compared with the corresponding character in the other string.

Note that the not-equal values are specified as a range. If the first parameter is less than the second parameter, the value can be any negative value. Likewise, if the first parameter is greater than the second parameter, the value can be any positive number. This differs from other situations, such as EOF, where we can rely on one given value being returned.

[5] Wide-character functions are available. Their function declarations are found in Appendix F.

The string compare operation is shown in Figure 11-17.

strcmp

The function declaration for the string compare function is shown below.

```
int strcmp (const char* str1, const char* str2);
```

While the string compare functions return integer values, (<=1, 0, >=1), we can code a logical expression that returns true or false by using the compare function and relational operators. For example, to compare two strings for equal, we must write the statement as shown below.

```
if (strcmp(str1, str2) == 0)
   // strings are equal
else
   // strings are not equal
```

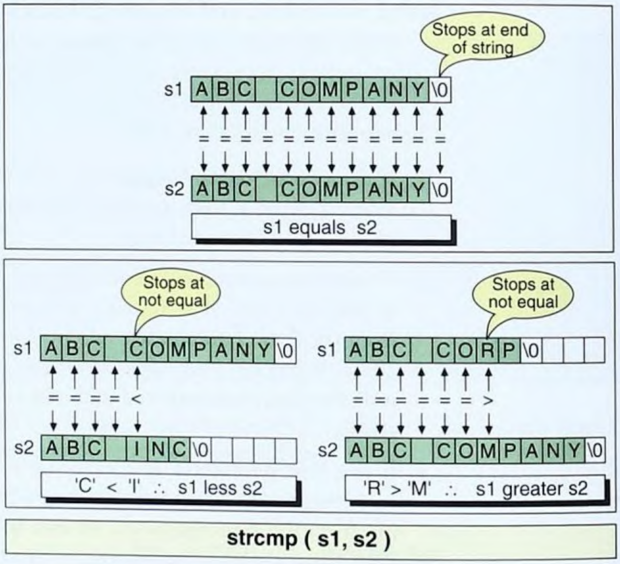


FIGURE 11-17 String Compares

The following statement tests whether the first string is less than the second string.

```
if (strcmp (string1, string2) < 0)
   // string1 is less than string2
```

To test for string1 greater than string2, use the following code:

```
if (strcmp (string1, string2) > 0)
   // string1 is greater than string2
```

We can also test for greater than or equal to with the following statement:

```
if (strcmp (string1, string2) >= 0)
   // string1 is greater than or equal to string2
```

strncmp

The string number compare function, *strncmp*, tests two strings for a specified maximum number of characters (size). The function declarations are shown below.

```
int strncmp (const char* str1, const char* str2,
                             int size);
```

In this function, size specifies the maximum number of characters to be compared in the first string. The *strncmp* compare logic is the same as seen in *strcmp*, except for the length limit. Table 11-1 shows the results of comparing two strings using the *strncmp* function for various sizes.

string1	string2	Size	Results	Returns
"ABC123"	"ABC123"	8	equal	0
"ABC123"	"ABC456"	3	equal	0
"ABC123"	"ABC456"	4	string1 < string2	< 0
"ABC123"	"ABC"	3	equal	0
"ABC123"	"ABC"	4	string1 > string2	> 0
"ABC"	"ABC123"	3	equal	0
"ABC123"	"123ABC"	–1	equal	0

TABLE 11-1 Results for String Compare

String Concatenate

The string concatenate functions append one string to the end of another. They return the address pointers to the destination string. The size of the destination string array is assumed to be large enough to hold the resulting string. If it isn't, the data at the end of the string array are destroyed. As we saw with string copy, the results are unpredictable if the strings overlap.

Basic String Concatenation

The function declaration for string concatenation, *strcat*, is shown below.

```
char* strcat (char* str1, const char* str2);
```

The function copies `str2` to the end of `str1`, beginning with `str1`'s delimiter. That is, the delimiter is replaced with the first character of `str2`. The delimiter from `str2` is copied to the resulting string to ensure that a valid string results. The length of the resulting string is the sum of the length of `str1` plus the length of `str2`. Figure 11-18a shows the operation.

String Concatenation—Length Controlled

The function declaration for length-controlled string concatenate function, *strncat*, is shown below.

```
char* strncat (char* str1, const char* str2, int size);
```

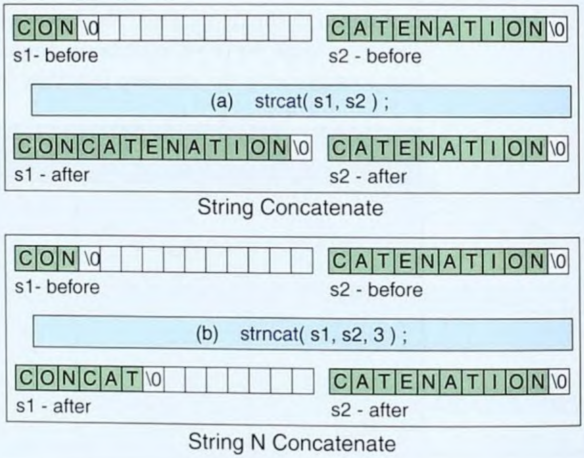


FIGURE 11-18 String Concatenation

Figure 11-18b demonstrates the operation of the `strncat` function. If the length of `string2` is less than `size`, then the call works the same as the basic string concatenation described above. However, if the length of `string2` is greater than `size`, then only the number of characters specified by `size` are copied, and a null character is appended at the end.

If the value of `size` is zero or less than zero, then both strings are treated as null, and no characters are moved. The variable `string1` is unchanged.

Character in String

Sometimes we need to find the location of a character in a string. Two string functions search for a character in a string. The first function is called string character *strchr*; it searches for the first occurrence of a character from the beginning of the string. The second string rear character called *strrchr*, searches for the first occurrence beginning at the rear and working toward the front.

In either case, if the character is located, the function returns a pointer to it. If the character is not in the string, the function returns a null pointer. The declarations for these functions are shown below.

```
char* strchr  (const char* string, int ch);
char* strrchr (const char* string, int ch);
```

Note that, as often is the case in the library functions, the character in the ASCII functions is typed as an integer. We don't need to worry about this. The compiler will implicitly cast any character types to integer types before the call. Figure 11-19 shows three examples of character-in-string calls.

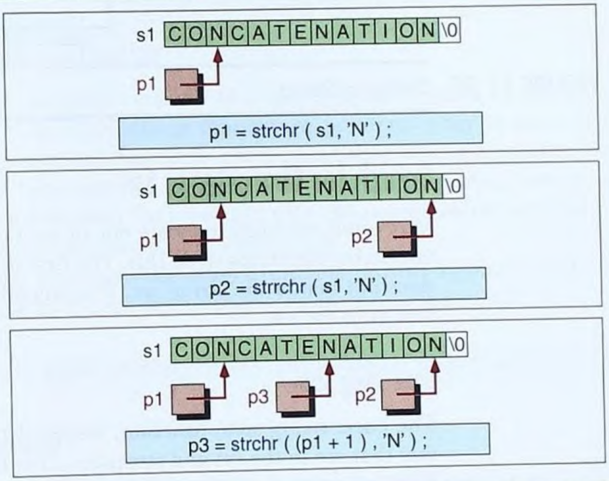

s1
C O N C A T E N A T I O N \0
p1
p1 = strchr (s1, 'N') ;
s1
C O N C A T E N A T I O N \0
p1
p2
p2 = strrchr (s1, 'N') ;
s1
C O N C A T E N A T I O N \0
p1
p3
p2
p3 = strchr ((p1 + 1) , 'N') ;

FIGURE 11-19 Character in String (*Strchr*)

In the first example, we locate the first N at the third position in the string (`s1[2]`). The pointer is then assigned to `p1`. In the second example, we locate the last N in the string at `s1[12]`. In the third example, we want to locate the second N. To do this, we need to start the search after the first N. Since we saved the location of the first N, this is easily done with a pointer and pointer arithmetic as shown.

Search for a Substring

If we can locate a character in a string, we should be able to locate a string in a string. We can, but only from the beginning of the string. There is no function to locate a substring starting at the rear. The function declaration for *strstr* is shown below.

```
char* strstr (const char* string,
              const char* sub_string);
```

As indicated by the function declaration, this function also returns a pointer to a character. This pointer identifies the beginning of the substring in the string. If the substring does not exist, then they return a null pointer. Figure 11-20 demonstrates the operation of string in string.

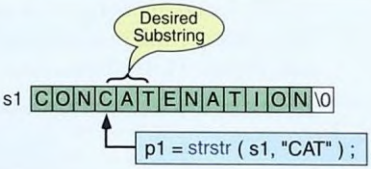


FIGURE 11-20 String in String

Search for Character in Set

Very often, we need to locate one of a set of characters in a string. C provides two string functions to do this. The first *strspn*, locates the first character that does *not* match the string set. The second *strcspn*, locates the first character that is in the set.

Basic String Span

The basic string span function, *strspn*, searches the string, spanning characters that are in the set and stopping at the first character that is not in the set. They return the number of characters that matched those in the set. If no characters match those in the set, they return zero. The function declaration is seen below.

```
int strspn(const char*     str, const char*     set);
```

An example of string span is shown in Figure 11-21. We use *strspn* to determine the number of characters that match the characters in the string set. In this example, `len` is set to five, since the first five characters match the set.

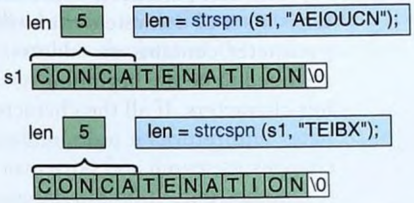


FIGURE 11-21 String Span

Complemented String Span

The second function, *strcspn*, is string *complement* span; its functions stop at the first character that matches one of the characters in the set. If none of the characters in the string match the set, they return the length of the string. Figure 11-21 also contains an example of *strcspn*. The function declarations are shown below.

```
int strcspn(const char* str, const char* set);
```

String Span—Pointer

Very often, we need a pointer to a substring rather than its index location. C provides a set of functions that return pointers. They operate just like the string complement functions except that they return a pointer to the first character that matches the set. The *p* in the function names stands for pointer—*brk* is short for break. As with the complement spans, if no matching characters are found, they return null.

Similar to *strcspn*, the *strpbrk* function returns a pointer to the first character found in the set. Its function declaration is shown below.[6]

```
char* strpbrk (const char* str, const char* set);
```

String Token

The string token function, *strtok*, is used to locate substrings, called tokens, in a string. Its most common use is to parse a string into tokens, much as a compiler parses lines of code. Depending on how it is called, it either locates the first or the next token in a string. Its function declaration statement is shown below.

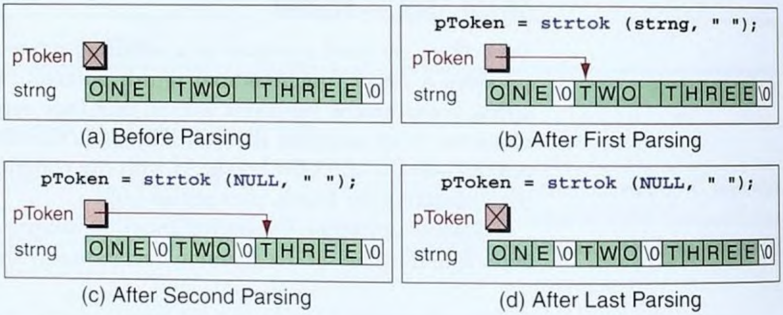
```
char* strtok (char* str, const char* delimiters);
```

[6] Traditional C has a function called *strrpbrk* that returns a pointer to the last character found in the set. This function is not part of the ANSI/ISO C function library.

The first parameter is the string that is being parsed; the second parameter is a set of delimiters that will be used to parse the first string. If the first parameter contains an address, then *strtok* starts at that address, which is assumed to be the beginning of the string. It first skips over all leading delimiter characters. If all the characters in the string are delimiters, then it terminates and returns a null pointer. When it finds a nondelimiter character, it changes its search and skips over all characters that are not in the set; that is, it searches until it finds a delimiter. When a delimiter is found, it is changed to a null character (`'\0'`), which turns the token just parsed into a string.

If the first parameter is not a string, *strtok* assumes that it has already parsed part of the string and begins looking at the end of the previous string token for the next delimiter. When a delimiter is located, it again changes the delimiter to a null character, marking the end of the token, and returns a pointer to the new token string.

Let's look at a simple example of a string containing words separated by spaces. We begin by calling string token with the address of the full string. It returns the address of the first character of the first string that was just parsed. The second execution of the string token function parses the second string and returns its address, and so on until the complete string has been parsed. This design is shown in Figure 11-22.



```
pToken = strtok (strng, " ");
pToken = strtok (NULL, " ");
pToken = strtok (NULL, " ");
```

pToken
strng
O N E T W O T H R E E \0
O N E \0 T W O T H R E E \0
O N E \0 T W O \0 T H R E E \0
O N E \0 T W O \0 T H R E E \0

(a) Before Parsing

(b) After First Parsing

(c) After Second Parsing

(d) After Last Parsing

FIGURE 11-22 Parse a Simple String

String to Number

The standard library (*stdlib.h*) library provides several functions that convert a string to a number. The two most important are string to long and string to double. A complete list is found at the end of this section.

String to Long

The string-to-long function, *strtol*, converts a string to a long integer.[7] It skips leading whitespace characters and stops with the first non-numeric character,

[7] To convert a string to an unsigned long integer, use *strtoul*.

which is considered to be the start of a trailing string. The address of the trailing string is stored in the second parameter, unless it is a null pointer. A third parameter determines the base of the string number. The function declaration is shown below.

```
long strtol (char* str, char** ptr, int base);
```

The base is determined by the following rules:

1. The base may be 0 or 2…35.
2. The letters a…z or A…Z represent the values 10…35. Only the numeric and alphabetic characters less than the base are permitted in any string.
3. If the base is 0, the format is determined by the string as follows:
   a. If the number begins `0x` or `0X`, the number is a hexadecimal constant.
   b. If the first digit is 0 and the second digit is not `x` or `X`, the number is an octal constant.
   c. If the first digit is a nonzero, the number is a decimal constant.

If the string does not begin with a valid number, zero is returned and the trailing string pointer is set to the beginning of the string. Program 11-13 demonstrates the use of *strtol*.

PROGRAM 11-13 Demonstrate String to Long

```
/* Demonstrate string to long function.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
// Local Declarations
   long  num;
   char* ptr;

// Statements
   num = strtol ("12345 Decimal constant: ", &ptr, 0);
   printf("%s %ld\n", ptr, num);

   num = strtol ("11001 Binary constant : ", &ptr, 2);
   printf("%s %ld\n", ptr, num);

   num = strtol ("13572 Octal constant  : ", &ptr, 8);
   printf("%s %ld\n", ptr, num);
```

continued

PROGRAM 11-13 Demonstrate String to Long *(continued)*

```

   num = strtol (" 7AbC Hex constant    : ", &ptr, 16);
   printf("%s %ld\n", ptr, num);

   num = strtol ("11001 Base 0-Decimal  : ", &ptr, 0);
   printf("%s %ld\n", ptr, num);

   num = strtol ("01101 Base 0-Octal    : ", &ptr, 0);
   printf("%s %ld\n", ptr, num);

   num = strtol ("0x7AbC Base 0-Hex      : ", &ptr, 0);
   printf("%s %ld\n", ptr, num);

   num = strtol ("Invalid input    : ", &ptr, 0);
   printf("%s %ld\n", ptr, num);

   return 0;
}  // main
```

```
Results:
   Decimal constant:  12345
   Binary constant :  25
   Octal constant  :  6010
   Hex constant    :  31420
   Base 0-Decimal  :  11001
   Base 0-Octal    :  577
   Base 0-Hex      :  31420
   Invalid input   :  0
```

Program 11-13 Analysis The first thing to note in this program is that the address of the character pointer must be passed to *strtol*. If this surprises you, go back and look at the *strtol* function declaration statement again. Note that the declaration of `ptr` has two asterisks.

Observe that the pointer is set to the first character after the last valid digit. In our example, this character was always a space. If there were no characters after the last digit, the pointer would point to the string delimiter, resulting in a null string.

String to Double

The string-to-double functions—*strtod* and *wcstod*—are similar to the string-to-long functions, except that they do not have a base. The function declaration is seen below.

```
double strtod (char* str, char** ptr);
```

The numeric string must be a floating-point in either decimal or scientific notation. The valid numeric values in the string are the digits, the plus and minus signs, and the letter *e* or *E* in the exponent. After using the longest possible sequence of digits and characters to convert the string to a number, *strtod* sets the pointer to the address of the next character in the string. If no characters are present, the pointer is set to the string delimiter. As with string-to-long, if the number is invalid, zero is returned and the pointer is set to the beginning of the string.

Other Number Functions

Table 11-2 lists the complete set of string-to-number functions.

Numeric Format	ASCII Function	Wide-character Function
double	strtod	wcstod
float	strtof	wcstof
long double	strtold	wcstold
long int	strtol	wcstol
long long int	strtoll	wcstoll
unsigned long int	strtoul	wcstoul
unsigned long long int	strtoull	wcstoull

TABLE 11-2 String-to-Number Functions

String Examples

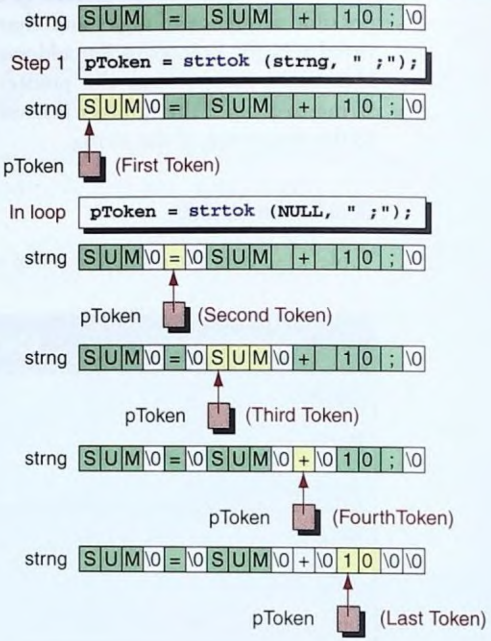
In this section, we discuss two functions that use string functions. The first uses *strtok* to parse an algebraic expression. The second uses string input/output functions and several string manipulation functions to compare strings.

EXAMPLE 11-11 Parsing Tokens

Assume we want to parse a string containing a simple algebraic expression as shown below.

```
sum = sum + 10;
```

Each token in this simple expression is separated by whitespace. Our program will identify each token and print it out. The step-by-step parsing is shown in Figure 11-23. The code is shown in Program 11-14.


strng
S U M = S U M + 1 0 ; \0
Step 1
strng
S U M \0 = S U M + 1 0 ; \0
pToken
(First Token)
In loop
strng
S U M \0 = \0 S U M + 1 0 ; \0
pToken
(Second Token)
strng
S U M \0 = \0 S U M \0 + 1 0 ; \0
pToken
(Third Token)
strng
S U M \0 = \0 S U M \0 + \0 1 0 ; \0
pToken
(FourthToken)
strng
S U M \0 = \0 S U M \0 + \0 1 0 \0 \0
pToken
(Last Token)

```
pToken = strtok (strng, " ;");
```

```
pToken = strtok (NULL, " ;");
```

FIGURE 11-23 Parsing with String Token

PROGRAM 11-14 Parsing a String with String Token

```
/* Parse a simple algebraic expression.
      Written by:
      Date:
*/
#include <stdio.h>
#include <string.h>

int main (void)
{
// Local Declarations
   char  strng [16] = "sum = sum + 10;";
   char* pToken;
   int   tokenCount;

// Statements
   tokenCount = 0;
   pToken = strtok (strng, " ;");

```

continued

PROGRAM 11-14 Parsing a String with String Token *(continued)*

```c
   while (pToken)
      {
       tokenCount++;
       printf("Token %2d contains %s\n",
               tokenCount, pToken);
       pToken = strtok (NULL, " ;");
      } // while

   printf("\nEnd of tokens\n");
   return 0;
}  // main
```

```
Results:
Token  1 contains sum
Token  2 contains =
Token  3 contains sum
Token  4 contains +
Token  5 contains 10

End of tokens
```

Program 11-14 Analysis Since the first call to *strtok* must contain the address of the string, it is coded *before* the loop. If the string contains at least one token, the first call will return a valid address and the *while* loop will print it and parse out the remaining tokens. Note that the delimiter set includes the semicolon as well as the blank between tokens. The semicolon serves as the last token in the string.

Compare Packed Strings

When working with strings, we often find that two strings are logically the same but physically different to the computer. For example, consider a program that generates mailing labels. Often a name is put into a mailing list with an extra space or other character that prevents it from being matched to an existing name. One way to eliminate such errors is to compare only the letters of the names by removing everything except alphabetic characters. Program 11-15 shows a function that compares two strings after packing the data so that only letters are left.

PROGRAM 11-15 Compare Packed String Function

```c
/* This program packs and compares a string.
      Written by:
      Date:
*/
#include <stdio.h>
#include <string.h>
```

continued

PROGRAM 11-15 Compare Packed String Function *(continued)*

```

#define ALPHA \
 "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"

// Function Declarations
int  strCmpPk (char* S1, char* S2);
void strPk    (char* s1, char* s2);

int main (void)
{
// Local Declarations
   int  cmpResult;
   char s1[80];
   char s2[80];

// Statements
   printf("Please enter first string:\n");
   fgets (s1, 80, stdin);
   s1[strlen(s1) - 1] = '\0';

   printf("Please enter second string:\n");
   fgets (s2, 80, stdin);
   s2[strlen(s2) - 1] = '\0';

   cmpResult = strCmpPk (s1, s2);
   if (cmpResult < 0)
       printf("string1 < string2\n");
   else if (cmpResult > 0)
       printf("string1 > string2\n");
   else
       printf("string1 == string2\n");

   return 0;
}  // main

/* ================== strCmpPk ===================
   Packs two strings and then compares them.
      Pre   s1 and s2 contain strings
      Post  returns result of strcmp of packed strings
*/
int strCmpPk (char* s1, char* s2)
{
// Local Declarations
   char s1In [80];
```

continue

PROGRAM 11-15 Compare Packed String Function *(continued)*

```
   char s1Out[81];
   char s2In [80];
   char s2Out[81];

// Statements
   strncpy (s1In, s1, sizeof(s1In) - 1);
   strncpy (s2In, s2, sizeof(s2In) - 1);
   strPk (s1In, s1Out);
   strPk (s2In, s2Out);
   return (strcmp (s1Out, s2Out));
} // strCmpPk

/* =================== strPk ===================
   Deletes all non-alpha characters from s1 and
   copies to s2.
      Pre   s1 is a string
      Post  packed string in s2
            s1 destroyed
*/
void strPk (char* s1, char* s2)
{
// Local Declarations
   int strSize;

// Statements
   *s2 = '\0';
   while (*s1 != '\0')
      {
       // Find non-alpha character & replace
       strSize = strspn(s1, ALPHA);
       s1[strSize] = '\0';
       strncat (s2, s1, 79 - strlen(s2));
       s1 += strSize + 1;
      } // while
   return;
}  // strPk
```

```
Results:
Please enter first string:
a b!c 234d
Please enter second string:
abcd
string1 == string2
```

continued

PROGRAM 11-15 Compare Packed String Function *(continued)*

```
Please enter first string:
abcd
Please enter second string:
aabb
string1 > string2
```

Program 11-15 Analysis

To test the string compare, we need to write a test driver. Since the test driver is throw-away code, we have coded the test logic in *main*.

Statement 8 contains a C language construct that we have not used before: the *statement continuation*. When a statement does not fit on a line, it can be continued by putting an escape character (\) immediately before the end of the line. This is consistent with the meaning of the escape character; in this case, it means escape the end of line that follows; it is not really an end of line. This allows us to continue the *define* statement on the next line.

Since we are passing addresses to strings, we need to be careful that we don't destroy the original data. The first thing we do in `strCmpPk`, therefore, is to use *strncpy* to copy the input string to a work area. Once we have protected the original data, we pack the two strings to be compared and then use the *strcmp* function to determine if the reformatted data are equal.

The `strPk` function merits a little discussion. It uses a *while* loop to process the input string. Each iteration of the loop scans the valid characters, as defined in `ALPHA`, looking for invalid data. When `strPk` finds an invalid character—and there will always be at least one, the input string delimiter—it (1) replaces that character with a string delimiter, (2) concatenates the input string to the output string, and then (3) adjusts the input string pointer to one past the last scanned character. This is necessary to point it to the data beyond the string just copied.

But what if the pointer adjustment puts it beyond the end of the input data? In that case, we will have a delimiter, because we used *strncpy* to copy the data from the original input, and it pads the output string with delimiters. Therefore, the character after the original string delimiter is another delimiter. Note that we made the output area one character larger than the input area to provide for this extra delimiter even with a full string.

At least three test cases are required to test the program. We show two of them in the result. In addition to an equal and greater-than result, you should conduct at least one less-than test case.

11.6 String/Data Conversion

A common set of applications format data by either converting a sequence o characters into corresponding data types or vice versa. Two such application are parsing and telecommunications.

C already has an extensive set of data conversion functions created fo *scanf* and *printf*. The C standard also includes two sets of memory formattin functions, one for ASCII characters and one for wide characters, that use these same formatting functions. Rather than reading and write files, how ever, they "read" and "write" strings in memory.

String to Data Conversion

The string scan function is called *sscanf*. This function scans a string as though the data were coming from a file. Just like *fscanf*, it requires a format string to provide the formatting parameters for the data. All *fscanf* format codes are valid for the scan memory functions. Like *fscanf*, the scan memory functions also return the number of variables successfully formatted. If they reach the end of a string before all the format string conversion codes have been used—that is, if they attempt to "read" beyond the end of string—they return an end-of-file flag. The basic concept is shown in Figure 11-24.

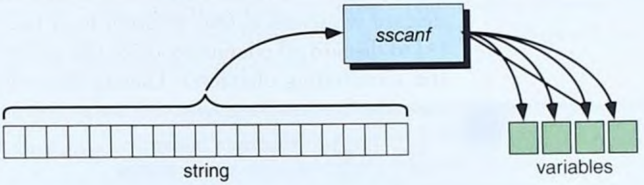


FIGURE 11-24 *sscanf* Operation

The function declaration is shown below.

```
int sscanf (char* str, const char* frmt_str, …);
```

The first parameter specifies the string holding the data to be scanned. The ellipsis (...) indicates that a variable number of pointers identify the fields into which the formatted data are to be placed.

Let's demonstrate the use of *sscanf* with an example. Assume that we have a string that contains a name terminated with a semicolon, a four-digit student number, an exam score, and a character grade. Each field is separated from the rest by at least one whitespace character. A data sample is shown below.

```
Einstein, Albert; 1234 97 A
```

In this problem, we want to format the name and student number as strings, the score as an integer, and the grade as a character. We therefore construct the following format string:

```
"%49[^;] %*c %4s %d %*[^ABCDF] %c"
```

Let's examine each format code individually. First, we are expecting a string of up to 49 characters terminated by a semicolon. Note that we use the scan set with a terminating token of a semicolon. Following the scan set that reads to a semicolon is a format code that reads and discards one character. The asterisk is a flag indicating that the character is not to be stored. This code is necessary because the terminating semicolon from the string remains in the input string. We need to read and discard it.

The third format code is a simple four-character string. Any leading whitespace is discarded with the string token (). Likewise, the third field, the score, is converted to an integer format after the leading whitespace is discarded.

Parsing the grade is a little more difficult. We can have one or more whitespaces between the numeric grade and the alphabetic grade, but we must dispose of them ourselves, because the character token (%c) does not discard whitespace. Our solution is to use a scan set with the suppress flag (*) to discard all characters up to the grade. Again, we use the negation to set the terminating character. Having discarded all the leading whitespace, we can now format the grade with a character token.

Using our example above, we can code the *sscanf* call as shown below.

```
sscanf(strIn, "%25[^;]%*c%4s%d%*[^ABCDF]%c",
          name, stuNo, &score, &grade);
```

Note that this code has six format codes but only four variables because the second and fifth fields have their formatting suppressed.

sscanf **is a one-to-many function.**

It splits one string into many variables.

Data to String Conversion

The string print function, *sprintf*, follows the rules of *fprintf*. Rather than sending the data to a file, however, it simply "writes" them to a string. When all data have been formatted to the string, a terminating null character is added to make the result a valid string. If an error is detected, *sprintf* returns any negative value, traditionally EOF. If the formatting is successful, it returns the number of characters formatted, not counting the terminating null character. The string print operation is shown in Figure 11-25.

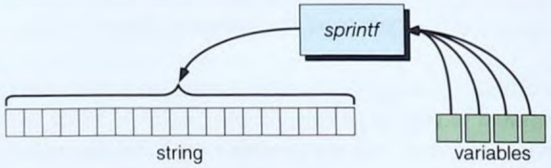


FIGURE 11-25 *sprintf* Operation

The function declaration for formatting memory strings is shown below.

```
int sprintf  (       char* out_string,
               const char* format_string, …);
```

The first parameter is a pointer to a string that will contain the formatted output. The format string is the same as *printf* and follows all of the same rules. The ellipse (…) contains the fields that correspond to the format codes in the format string.

It is our responsibility to ensure that the output string is large enough to hold all the formatted data. If it is not large enough, *sprintf* destroys memory contents after the output string variable. The wide-character function, *swprintf*, on the other hand limits the output. The additional parameter, `max_out`, specifies the maximum wide characters.

sprintf is a many-to-one function.

It joins many pieces of data into one string.

EXAMPLE 11-12 Demonstrate Memory Formatting

To demonstrate the memory formatting functions, we first format a string to variables and print them. We then format them back to a string and print the string. The data use the format described in "String to Data Conversion." The code is seen in Program 11-16.

PROGRAM 11-16 Demonstrate Memory Formatting

```
/* Demonstrate memory formatting.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   char  strng[80] = "Einstein, Albert; 1234 97 A";
   char  strngOut[80];
   char  name[50];
   char  id[5];
   int   score;
   char  grade;

// Statements
   printf("String contains:  \"%s\"\n", strng);
```

continued

PROGRAM 11-16 Demonstrate Memory Formatting *(continued)*

```

  sscanf(strng, "%49[^;] %*c %4s %d %c",
         name, id, &score, &grade);

  printf("Reformatted data:  \n");
  printf("   Name:           \"%s\"\n", name);
  printf("   id:             \"%s\"\n", id);
  printf("   score:           %d\n", score);
  printf("   grade:           %c\n", grade);

  sprintf(strngOut, "%s %4s %3d %c",
            name, id, score, grade);
    printf("New string:       \"%s\"\n", strngOut);
  return 0;
} // main
```

```
Results:
String contains:  "Einstein, Albert; 1234 97 A"
Reformatted data:
   Name:          "Einstein, Albert"
   id:            "1234"
   score:          97
   grade:          A
New string:       "Einstein, Albert 1234  97 A"
```

EXAMPLE 11-13 Range of Internet Addresses

As a practical example, let's write a program that, given two Internet addresses, determines all of the addresses in their range. An Internet address is a 4-byte Dotted Decimal Notation address in the form X.Y.Z.T. Each byte can have a decimal value between 0 and 255.

To determine the range, we multiply each portion of the address by 256 and add it to a counter. The second address is then subtracted from the first, which gives the number of addresses in the range.

This program could be used by system administrators when their organization is given a range of addresses. For example, if an organization is given the range of address 193.78.64.0 to 193.78.66.255, the system administrator needs to know how many addresses are available for computers to be connected to the Internet. Program 11-17 answers this question.

PROGRAM 11-17 Determine Internet Addresses in a Range

```
/* Given two Internet addresses, determine the number
   of unique addresses in their range.
```

continued

PROGRAM 11-17 Determine Internet Addresses in a Range *(continued)*

```
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
// Local Declarations
   unsigned int   strt[4];
   unsigned int   end[4];
   unsigned long  add1 = 0;
   unsigned long  add2 = 0;
   unsigned long  range;

   char addr1[15];
   char addr2[15];

// Statements
   printf ("Enter first address:  ");
   fgets (addr1 , sizeof (addr1) , stdin);

   printf ("Enter second address: ");
   fgets (addr2 , sizeof (addr2) , stdin);

   sscanf (addr1 , "%d %*c %d %*c %d %*c %d\n",
           &strt[3], &strt[2], &strt[1], &strt[0]);
   sscanf (addr2 , "%d %*c %d %*c %d %*c %d\n",
           &end[3], &end[2], &end[1], &end[0]);

   for (int i = 3 ; i >= 0  ; i--)
       {
         add1 = add1 * 256 + strt[i];
         add2 = add2 * 256 + end[i];
       } // for
   range = abs (add1 - add2) + 1;

   printf ("\nFirst Address: %s", addr1);
   printf ("Second Address %s", addr2);
   printf ("\nThe range: %ld\n", range);

   return 0;
}  // main
```

continued

PROGRAM 11-17 Determine Internet Addresses in a Range *(continued)*

```
Results:
Enter first address:  23.56.34.0
Enter second address: 23.56.32.255

First Address: 23.56.34.0
Second Address 23.56.32.255

The range: 258
```

11.7 A Programming Example—Morse Code

Morse code, patented by Samuel F. B. Morse in 1837, is the language that was used to send messages by telegraph from the middle of the nineteenth century until the advent of the modern telephone and today's computer-controlled communications systems. In Morse code, each letter in the alphabet is represented by a series of dots and dashes, as shown in Table 11-3.

Letter	Code	Letter	Code	Letter	Code	Letter	Code
A	.-	H	....	O	---	V	...-
B	-...	I	..	P	.--.	W	.--
C	-.-.	J	.---	Q	--.-	X	-..-
D	-..	K	-.-	R	.-.	Y	-.--
E	.	L	.-..	S	...	Z	--..
F	..-.	M	--	T	-		
G	--.	N	-.	U	..-		

TABLE 11-3 Morse Code

Program 11-18 encodes (converts) a line of text to Morse code and decodes (converts) Morse code to a line of text. We use a two-dimensional array of pointers in which each row has two string pointers, one to a string containing English characters and one to a string containing the corresponding Morse code. Note that the pointers are stored in the program's memory, while the strings are stored in the heap. The array structure is shown in Figure 11-26.

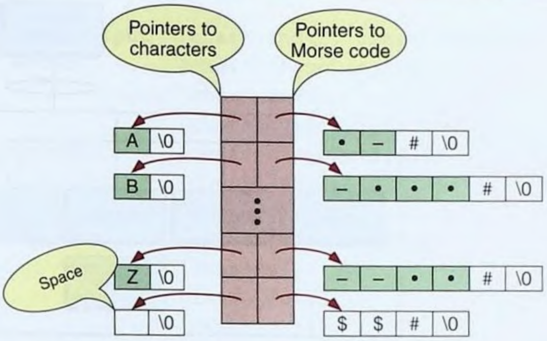


FIGURE 11-26 Character to Morse Code Structure

Each column has 27 pointers. Each pointer in the first column points to a string of length one, which contains one English letter (uppercase). Each pointer in the second column points to a string of varying size, a ragged array, which contains the corresponding Morse code for the English letter.

The program is menu driven. The menu, which is shown in Figure 11-27, has three options: encode English to Morse code, decode Morse code to English, and quit.

```
                MENU
          E            encode

          D            decode

          Q            quit
     Enter option: press return key:
```

FIGURE 11-27 Morse Code Menu

The program begins by initializing the conversion table. It then displays the menu and loops until the user quits. Each loop either encodes a line of English text or decodes a line of Morse code. The only allowable characters are the alphabetic characters and spaces. When Morse code is entered, each coded character is terminated by a pound sign (#). Morse code space is represented by two dollar signs ($$). The complete design is shown in Figure 11-28.

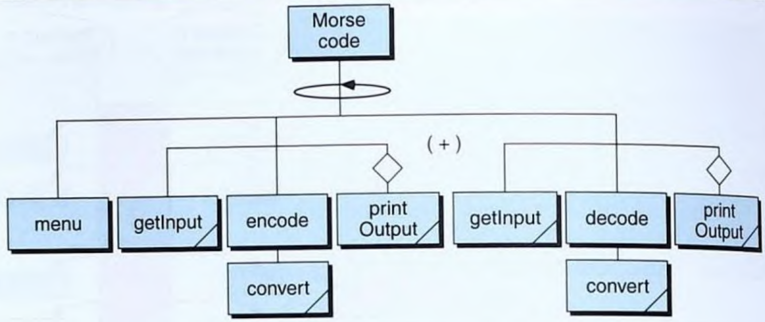


FIGURE 11-28 Morse Code Program Design

If the encode or decode function detects an error—such as an invalid character or an invalid Morse code sequence—it returns an error code, and the calling program prints an error message.

The solution, as shown in Program 11-18, uses three arrays. The first is the encode/decode (`encDec`) array. As seen in Figure 11-26, this is a two-dimensional array of pointers to English and Morse code values. The second array is an array of 81 characters to hold the input line. The third is an array of 81 characters to hold the output line. To ensure that we do not overrun the output line, we limit the input string to 16 characters when we are reading English text. When appropriate, we provide some analysis to points in the individual functions.

PROGRAM 11-18 Morse Code: *main*

```
/* Convert English to Morse or Morse to English.
      Written by:
      Date Written:
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <stdbool.h>

#define FLUSH while(getchar() != '\n')
#define STR_LEN 81

// Function Declarations
char menu          (void);
void getInput      (char* inStr);
```

continued

PROGRAM 11-18 Morse Code: *main (continued)*

```
void printOutput  (char* inStr, char* outSt);
bool encode       (char* (*encDec)[2],
                   char* inStr,
                   char* outStr);
bool decode       (char* (*encDec)[2],
                   char* inStr,
                   char* outStr);
int  convert      (char* (*encDec)[2],
                   char* s1,
                   int   col,
                   char* s2);

int main (void)
{
// Local Declarations
   char* encDec [27][2] =
      {
       { "A", ".-#" },
       { "B", "-...#" },
       { "C", "-.-.#" },
       { "D", "-..#" },
       { "E", ".#" },
       { "F", "..-.#" },
       { "G", "--.#" },
       { "H", "....#" },
       { "I", "..#" },
       { "J", ".---#" },
       { "K", "-.-#" },
       { "L", ".-..#" },
       { "M", "--#" },
       { "N", "-.#" },
       { "O", "---#" },
       { "P", ".--.#" },
       { "Q", "--.-#" },
       { "R", ".-.#" },
       { "S", "...#" },
       { "T", "-#" },
       { "U", "..-#" },
       { "V", "...-#" },
       { "W", ".--#" },
       { "X", "-..-#" },
       { "Y", "-.--#" },
       { "Z", "--..#" },
       { " ", "$$#" },
```

continued

PROGRAM 11-18 Morse Code: *main (continued)*

```
      }; // Encode / Decode array
   char inStr  [STR_LEN];
   char outStr [STR_LEN];
   char option;
   bool done = false;

// Statements
   while (!done)
      {
       option = menu ();
       switch (option)
          {
           case 'E' :
                  getInput (inStr);
                  if (!encode (encDec, inStr, outStr))
                     {
                      printf("Error! Try again");
                      break;
                     } // if
                  printOutput (inStr, outStr);
                  break;
           case 'D' : getInput (inStr);
                  if (!decode (encDec, inStr, outStr))
                     {
                      printf("Error! Try again");
                      break;
                     } // if
                  printOutput (inStr, outStr);
                  break;
           default :
                  done = true;
                  printf("\nEnd of Morse Code.\n");
                  break;
          } // switch
      } // while
   return 0;
 }  // main
```

Program 11-18 Analysis The function *main* is rather straightforward. Although it could be argued that the *switch* statements should be in a subfunction, we place it here because it controls the entire program. One noteworthy point is the default condition. We test for only two options, encode and decode. If it is neither, then we assume quit. We can do this because we validate the option in `menu` (see Program 11-19). At this point in the program, therefore, `option` can be one of only three values. We test for two and default the third.

PROGRAM 11-19 Morse Code: Menu

```
/* ==================== menu =====================
   Display menu of choices; return selected character.
      Pre   nothing
      Post  returns validated option code
*/
char menu (void)
{
// Local Declarations
   char  option;
   bool  validData;

// Statements
   printf("\t\t\tM E N U \n");
   printf("\t\tE)  encode \n");
   printf("\t\tD)  decode \n");
   printf("\t\tQ)  quit \n");

   do
      {
       printf ("\nEnter option: press return key: ");
       option = toupper (getchar());
       FLUSH;
       if (option == 'E' || option == 'D' ||
           option == 'Q')
          validData = true;
       else
          {
           validData = false;
           printf("\aEnter only one option\n");
           printf(" \tE, D, or Q\n ");
          } // else
      } while (!validData);
   return option;
}  // menu
```

Program 11-19 Analysis The menu function in Program 11-19 displays the options and reads the user's choice. It then validates the choice and if invalid, displays an error message and asks for the option again. Although it is always good design to validate the user input, it must be validated in the correct place. Since `menu` is communicating with the user, this is the logical place to do the validation. One more point: As the option is read, it is converted to uppercase. Not only does this make the validation simpler, but it also simplifies the *switch* statement in *main*. Whenever you have a single character code, convert it to uppercase or lowercase for processing in the program.

The input string is read in Program 11-20.

PROGRAM 11-20 Morse Code: Get Input

```
/* =================== getInput ===================
   Reads input string to be encoded or decoded.
      Pre   inStr is a pointer to the input area
      Post  string read into input area
*/
void getInput (char* inStr)
{
// Statements
   printf ("\nEnter line of text to be coded: \n");
   fgets  (inStr, STR_LEN, stdin);

   // Eliminate newline in input string
   *(inStr-1 + strlen(inStr)) = '\0';

   if (isalpha(*inStr) && strlen(inStr) > 16)
       {
        // Exceeds English input length
        printf("\n***WARNING: Input length exceeded: ");
        printf("Only 16 chars will be encoded.\a\a\n");
        *(inStr + 16) = '\0';
       } // if
    return;
 }  // getInput
```

Program 11-20 Analysis Note that to prevent a runaway string in `getInput` we use the *fgets* function and specify the maximum number of characters in the string area. This function creates a minor problem, however; it places the newline in the input string. We must therefore overlay it with a null character.

The code to print the output of this program is shown in Program 11-21.

PROGRAM 11-21 Morse Code: Print Output

```
/* ================= printOutput =================
   Print the input and the transformed output
      Pre   inStr contains the input data
            outStr contains the transformed string
      Post  output printed
*/
void printOutput (char* inStr, char* outStr)
{
// Statements
   printf("\nThe information entered was: \n");
   puts(inStr);
   printf("\nThe transformed information is: \n");
```

continued

PROGRAM 11-21 Morse Code: Print Output *(continued)*

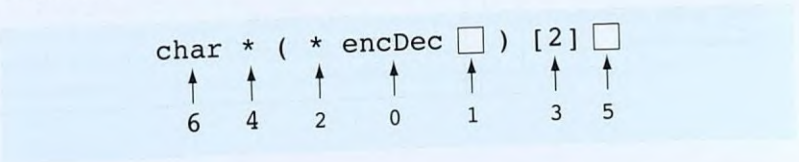
```
   puts(outStr);
   return;
}  // printOutput
```

Program 11-22 transforms character data to Morse code.

PROGRAM 11-22 Morse Code: Encode to Morse

```
 1  /* =================== encode ====================
 2     Transforms character data to Morse code
 3        Pre    encDec is the conversion table
 4               inStr contains data to be put into Morse
 5        Post   data have been encoded in outStr
 6        Return true  if all valid characters;
 7               false if invalid character found
 8  */
 9  bool encode (char* (*encDec)[2],
10               char* inStr, char* outStr)
11  {
12  // Local Declarations
13     char s1[2];
14     char s2[6];
15     int  error = 0;
16
17  // Statements
18     outStr[0] = '\0';
19     while (*inStr != '\0' && !error)
20        {
21         s1[0] = toupper(*inStr);
22         s1[1] = '\0';
23         error = !convert (encDec, s1, 0, s2);
24         strcat (outStr, s2);
25         inStr++;
26        } // while
27      return (!error);
28  }  // encode
```

Program 11-22 Analysis Note how the encode/decode table is declared in the formal parameter list in statement 9. It uses a complex type. To understand it, let's use right–left analysis (see Chapter 10).



Reading this declaration, at 0 we have `encDec`. Moving to the right, we find the 1, which is ignored. Moving left, at 2 we find a pointer. We therefore have `encDec` as a pointer. Moving right to 3, we see that it is a pointer to an array of two. Moving to the left to 4, we see that it is an array of pointers.

At this point, we see that `encDec` is a pointer to an array of two pointers. Now, moving back to the right, we see that 5 is empty, so we go to the left and find *char* at 6. The final type is therefore a pointer to an array of two pointers to *char*. Referring back to Figure 11-26, we see that this is exactly what we have.

The code to decode Morse code to English is shown in Program 11-23.

PROGRAM 11-23 Morse code: Decode to English

```
/* ==================== decode =====================
   Transforms Morse code data to character string
      Pre    encDec is the conversion table
             inStr contains data to transform to string
      Post   data encoded and placed in outStr
      Return true  if all valid characters;
             false if invalid character found
*/
bool decode (char* (*encDec)[2],
             char* inStr, char* outStr)
{
// Local Declarations
   char  s1[6];
   char  s2[2];
   bool  error = false;
   int   i;

// Statements
   outStr[0] ='\0';
   while (*inStr != '\0' && !error)
      {
       for (i = 0; i < 5 && *inStr != '#'; i++, inStr++)
           s1[i] = *inStr;

       s1[i]   = *inStr;
       s1[++i] = '\0';

       error = !convert (encDec, s1, 1, s2);
       strcat (outStr, s2);
       inStr++;
      } // while
    return (!error);
} // decode
```

Program 11-24 converts characters to Morse code and Morse code to characters.

PROGRAM 11-24 Morse Code: Convert Codes

```
 1  /* ==================== convert =====================
 2     Looks up code and converts to opposite format
 3        Pre   encDec is a pointer decoding table
 4              s1 is string being converted
 5              s2 is output string
 6               col is code: 0 for character to Morse
 7                            1 for Morse to character
 8        Post  converted output s2
 9  */
10  int convert (char* (*encDec)[2],
11               char* s1, int col, char* s2)
12  {
13  // Local Declarations
14     bool found = false;
15     int  i;
16
17  // Statements
18     for (i = 0; i < 27 && !found; i++)
19        found = !strcmp(s1, encDec[i][col]);
20
21     if (found)
22        strcpy (s2, encDec [i - 1][(col + 1) % 2]);
23     else
24        *s2 = '\0';
25
26     return found;
27  }  // convert
28  // ================ End of Program ================
```

rogram 11-24 Analysis The convert function does the actual conversion. Note, however, how it is designed to handle the conversion both from English to Morse and Morse to English. The only difference between the two conversions is which pointer we want to use. We pass the column to be used for the search as a parameter. Once we have located the correct string, we use the formula shown below to pick up the matching string (see statement 22).

```
(col + 1) % 2
```

If we are searching on the English letters in column 0, then the modulus of the column (0 + 1) is column 1, which contains the pointer to the matching Morse code string. Conversely, if we are searching Morse code using the pointers in column 1, then the modulus of the column (1 + 1) is column 0, which contains the pointer to the English letter.

11.8 Software Engineering

In this section, we've formalized some of the principles of good programming that we've discussed throughout the text. Although you will find little in this discussion of software engineering that relates directly to the subject of strings, all of the string functions have been written using the principles discussed on the following pages.

Program Design Concepts

You will study many different analysis and design tools as you advance in computer science. Since this text deals primarily with program analysis and design, we will discuss the primary tool we have used throughout the text: the structure chart.

The overriding premise of good design is that the program is modular; that is, it is well structured. This is the *sine qua non* of good programming. A program's degree of good design can be measured by two principles: Its modules are independent—that is, their implementation is hidden from the user—and they have a single purpose.

Information Hiding

Information hiding is the principle of program design in which the data structure and functional implementation are screened from the user. Modules are independent when their communication is only through well-defined parameters and their implementation is hidden from the user. The purpose of the function should be defined in its inputs and outputs, and the user should not need to know how it is implemented or how its data are structured. Pressman states it well when he says:

> Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. ... Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module.[8]

The concept of information hiding is the basis of the object-oriented design and programming. When you study data structures, you will see another technique used for information hiding, the abstract data type.

Cohesion

The most common weakness of function design is combining related processes into one primitive function. In Chapter 4 we discussed the concept

[8] Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 2nd ed, McGraw-Hill Series in Software Engineering and Technology (New York: McGraw-Hill, 1982), p. 228.

that each module (function) should do only one thing. This principle of structured programming is known as cohesion. **Cohesion**, first discussed by Larry Constantine in the 1960s,[9] is a measure of how closely the processes in a function are related.

We are concerned with cohesion for three primary reasons. The first and most important is *accuracy*. The more cohesive a function is, the simpler it is. The simpler it is, the easier it is to write, and the fewer errors it will have.

This is closely related to the second reason for high cohesion, *maintainability*. If a function is easy to understand, it is easy to change. This means that we will get the job done faster and with fewer errors.

Finally, cohesive modules are more *reusable*. Reusability is also closely related to the concepts of accuracy and ease of use. Existing functions have stood the test of time and have been tempered in the heat of use. They are more likely to be error free, and they certainly are easier and faster to develop.

Cohesion is most applicable to the primitive functions in a program, those that are at the bottom of the structure chart and less applicable to the controlling functions that appear above the lowest level. This does not mean, however, that cohesion can be ignored at the higher levels. To make the point with an absurd example, you wouldn't write a program to manage your checkbook and maintain your daily calendar. Even though both of these processes are related to things you do, they are so unrelated that you wouldn't put them in the same program. The same concept applies in a program. For example, at the lower levels of your program, you shouldn't combine functions that read data with functions that print a report.

The seven levels of cohesion are shown in Figure 11-29.

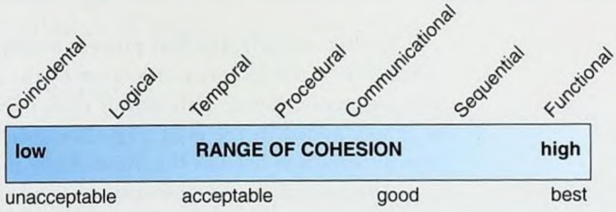


FIGURE 11-29 Types of Cohesion

Functional Cohesion

Functional modules that contain only one process demonstrate functional cohesion. This is the highest level of cohesion and the level that we should hold up as a model. Using the example of printing a report, the report function should call three lower-level functions, one to get the data, one to format

[9] E. N. Yourdon and L. L. Constantine, *Structured Design* (Upper Saddle River, N.J.: Prentice Hall, 1978).

and print the report header, and one to format and print the data. This design is shown in Figure 11-30. The print report heading function is optional because it is called only when a page is full.

Sequential Cohesion

A sequential module contains two or more related tasks that are closely tied together, usually with the output of one flowing as input to the other. An example of **sequential cohesion** is shown in the calculations for a sale. The design for this function might well be

1. Extend item prices
2. Sum items
3. Calculate sales tax
4. Calculate total

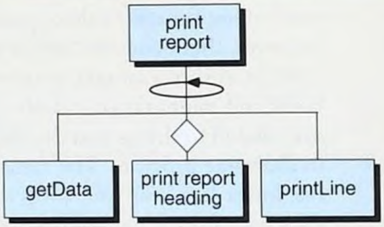


FIGURE 11-30 Example of Functional Cohesion

In this example, the first process multiplies the quantity purchased by the price. The extended prices are used by the process that calculates the sum of the purchased items. This sum is then used to calculate the sales tax, which is finally added to the sum to get the sale total. In each case, the output of one process was used as the input of the next process.

Although it is quite common to find the detail code for these processes combined into a single function, it does make the function more complex and less reusable. On the other hand, reusability would be a concern if the same or similar calculations were being made in different parts of one program.

Communicational Cohesion

Communicational cohesion combines processes that work on the same data. It is natural to have communicational cohesion in the higher modules in a program, but you should never find it at the primitive level. For example, consider a function that reads an inventory file, prints the current status of the parts, and then checks to see if any parts need to be ordered. The pseudocode for this process is shown in Algorithm 11-1.

ALGORITHM 11-1 Process Inventory Pseudocode

```
Algorithm Process Inventory
1 while not end of file
   1  read a record
   2  print report
   3  check reorder point
2 end while
end Process Inventory
```

All three of these processes use the same data. If they are calls to lower-level functions, they are acceptable. If the detail code is found in the function, however, the design is unacceptable.

The first three levels of cohesion are all considered to be good structured programming principles. Beyond this point, however, ease of understanding and implementation, maintainability, and accuracy begin to drop off rapidly. The next two levels should be used only at the higher levels of a structure chart, and then only rarely.

Well-structured programs are highly cohesive and loosely coupled.

Procedural Cohesion

The fourth level of cohesion, **procedural cohesion**, combines unrelated processes that are linked by a control flow. (This differs from sequential cohesion, where data flows from one process to the next.) As an example, consider the high-level logic main line of a program that builds and processes a list. A procedural flow could look like Algorithm 11-2.

ALGORITHM 11-2 Process List Pseudocode

```
Algorithm Process List
1 open files
2 initialize work areas
3 create list
4 print menu
5 while not stop
   1  get users response
   2  if locate …
   3  if insert …
   4  if delete …
   5  print menu
6 end while
7 clean up
8 close files
end Process List
```

A much better approach would be to have only three function calls in *main*, initialize, process, and end. Not only is this easier to understand, but it also simplifies the communication.

Temporal Cohesion

The fifth level, **temporal cohesion**, is acceptable only over a limited range of processes. It combines unrelated processes that always occur together. Two temporally cohesive functions are initialization and end of job. They are acceptable because they are used only once in the program and because they are never portable. Recognize, however, that they should still contain calls to functionally cohesive primitive functions whenever practical.

Logical and Coincidental Cohesion

The last two levels are seldom found in programs today. **Logical cohesion** combines processes that are related only by the entity that controls them. A function that conditionally opened different sets of files based on a flag passed as a parameter would be logically cohesive. Finally, **coincidental cohesion** combines processes that are unrelated. Coincidental cohesion exists only in theory. We have never seen a productional program that contained coincidental cohesion.

Summary

We have discussed two design concepts in this chapter: information hiding and cohesion. Cohesion describes the relationship among processes within a function. Keep functions as highly cohesive as possible. When designing a program, pay attention to the levels of cohesion. It is much easier to design high cohesion into a program than it is to program it in. For more information about these concepts, refer to *The Practical Guide to Structured Systems Design* by Meilir Page-Jones.[10]

[10] Meilir Page-Jones, *The Practical Guide to Structured Systems Design*, Yourdon Press Computing Series (Upper Saddle River, N.J.: Prentice-Hall, 1988).

11.9 Tips and Common Programming Errors

1. Do not confuse characters and string constants: The character constant is enclosed in single quotes, and the string constant is enclosed in double quotes.

2. Remember to allocate memory space for the string delimiter when declaring and defining an array of *char* to hold a string.

3. Strings are manipulated with string functions, not operators.

4. The header file *string.h* is required when using string functions.

5. The standard string functions require a delimited string. You cannot use them on an array of *char* that is not terminated with a delimiter.

6. Do not confuse string arrays and string pointers. In the following example, the first definition is a string (array of characters), and the second is a pointer to a character (string). Each is shown with a typical initialization statement.

```
char str[20];
strcpy (str, "Now is the time");
char* str;
str = "Now is the time";
```

7. Passing a character to a function when a string is required is another common error. This is most likely to occur with the formatted input/output functions, in which case, it is a logic error. Passing a character in place of a string when a function header is declared is a compile error.

8. Using the address operator for a parameter in the *scanf* function with a string is a coding error. The following is an invalid call and will most likely cause the program to fail:

```
scanf("%s", &string);                // Coding Error
```

9. It is a compile error to assign a string to a character, even when the character is a part of a string.

10. Using the assignment operator with strings instead of a function call is a compile error.

```
char string [20];
string = "This is an error";                // Compile Error
```

11. Since strings are built in an array structure, they may be accessed with indexes and pointers. When accessing individual bytes, it is a logic error to access beyond the end of the data structure (array).

11.10 Key Terms

cohesion
coincidental cohesion
communicational cohesion
delimited string
delimiter
scan set
fixed-length string
information hiding
length-controlled string
logical cohesion
procedural cohesion
sequential cohesion
temporal cohesion

11.11 Summary

- ❑ Strings can be fixed length or variable length.
- ❑ A variable-length string can be controlled by a length or a delimiter.
- ❑ The C language uses a null-terminated (delimited) variable-length string.
- ❑ A string constant in C is a sequence of characters enclosed in double quotes. A character constant is enclosed in single quotes.
- ❑ When string constants are used in a program, C automatically creates an array of characters, initializes it to a null-terminated string, and stores it, remembering its address.
- ❑ To store a string, we need an array of characters whose size is one more than the length of the string.
- ❑ There is a difference between an array of characters and a string. An array of characters is a derived data type (built on the type character) and is an intrinsic structure in C. A string is a data structure that uses the array as its basic type. It requires a delimiter for all of its operations.
- ❑ We can initialize a string:
    - ■ With an initializer when we define it.
    - ■ By using a string copy function
    - ■ By reading characters into it.
- ❑ A string identifier is a pointer constant to the first element in the string.
- ❑ An array of strings is a very efficient way of storing strings of different sizes.
- ❑ The *scanf* and *printf* functions can be used to read and write strings using the `%s` format.
- ❑ The scan set format, `%[...]`, can also be used in the *scanf* function to read a string.
- ❑ The functions *gets*, *fgets*, *puts*, and *fputs* are used for reading and writing strings. They transform a line to a string or a string to a line.

- ❑ The functions *sscanf* and *sprintf* are used for reading and writing strings from memory—that is, memory-to-memory formatting.
- ❑ The functions *strcpy* and *strncpy* are used to copy one string into another.
- ❑ The functions *strcmp* and *strncmp* are used to compare two strings.
- ❑ The function *strlen* is used to determine the length of a string.
- ❑ The functions *strcat* and *strncat* are used to concatenate one string to the end of the other.
- ❑ The functions *strchr* and *strrchr* are used to look for a character in a string.
- ❑ The function *strstr* is used to look for a substring in a string.
- ❑ The functions *strspn* and *strcspn* are used to locate the position of a set of characters in a string.
- ❑ Information hiding is the principle of programming in which the data structure and function's implementation are screened from the user.
- ❑ The cohesion principle dictates that each module must do only a single job.
- ❑ The *sine qua non* of good programming is modularity.
- ❑ Program design has seven layers of cohesion; only the first three (functional, sequential, and communicational) should be used for lower-level functions. The last two (logical and coincidental) should never be used.

11.12 Practice Sets

Review Questions

1. The two basic techniques for storing variable-length data are fixed-length strings and delimiters.

   a. True
   b. False

2. To initialize a string when it is defined, it is necessary to put the delimiter character before the terminating double quote, as in `"hello\0"`.

   a. True
   b. False

3. Both the *gets* and *fgets* functions change the newline to a delimiter.

   a. True
   b. False

4. The function that can be used to format several variables into one string in memory is string token (*strtok*).

   a. True
   b. False

5. A(n) ________________ is a series of characters treated as a unit.

   a. array
   b. character
   c. field
   d. record
   e. string

6. The delimiter in a C string is

   a. newline
   b. defined by the programmer
   c. a del character
   d. a null character
   e. not specified in ANSI C

7. Which of the following statements about string variables is false?

   a. The assignment operator copies the value of one string variable to another.
   b. The array bracket operator is not needed if a string is defined as a pointer to character.
   c. The string name is a pointer.
   d. When a string is initialized at definition time, C automatically inserts the delimiter.
   e. When a string is read with *scanf*, it automatically inserts the delimiter.

8. Which of the following *printf* statements will not print a string? (Assume that name is properly defined and formatted.)

   a. `printf("%s", name);`
   b. `printf("%30s", name);`
   c. `printf("%-30s", name);`
   d. `printf("%30[^\0]", name);`

9. The ________________ function reads a string from the standard input device.

   a. *fgets*
   b. *fputs*
   c. *gets*
   d. *getstr*
   e. *puts*

10. The ________________ string manipulation function returns the number of characters in the string excluding the null character.

    a. *strcmp*
    b. *strcpy*
    c. *strlen*
    d. *strsize*
    e. *strtok*

11. Which of the following statements determines if the contents of `string1` are the same as the contents of `string2`? Assume that `string1` and `string2` are properly defined and formatted strings.

    a. `if (string1 == string2)`
    b. `if (strcmp (string1, string2))`
    c. `if (strcmp (string1, string2) == 0)`
    d. `if (strcmp (string1, string2) < 0)`
    e. `if (strcmp (string1, string2) > 0)`

12. The ________________ string manipulation function adds a string to the end of another string.

    a. *stradd*
    b. *strcat*
    c. *strcmp*
    d. *strcpy*
    e. *strtok*

13. The ________________ string manipulation function determines if a character is contained in a string.

    a. *strchr*
    b. *strchar*
    c. *strcmp*
    d. *strcpy*
    e. *strtok*

14. The ________________ function is used to format a string into variables in memory.

    a. *scanf*
    b. *sscanf*
    c. *strfrmt*
    d. *strscan*
    e. *strtok*

15. The best level of cohesion is

    a. coincidental
    b. communicational
    c. procedural
    d. sequential
    e. functional

Exercises

16. Find the value of `*x`, `*(x + 1)`, and `*(x + 4)` for the following declaration:

```
char* x = "The life is beautiful";
```

17. Find the value of *y, *(y + 1), and *(y + 4) for the following program segment:

```
char x [] = "Life is beautiful";
char* y = &x [3];
```

18. What is the error in the following program block?

```
{
char* x;
scanf("%s", x);
}
```

19. What would be printed from the following program block?

```
{
 char  s1[50] = "xyzt";
 char* s2     = "xyAt";
 int   dif;
 dif = strcmp (s1, s2);
 printf("\n%d", dif);
}
```

20. What would be printed from the following program block?

```
{
 char  s1[50] = "xyzt";
 char* s2 = "uabefgnpanm";
 char* s3;
 char* s4;
 char* s5;
 char* s6;
 s3 = s1;
 s4 = s2;
 strcat (s1, s2);
 s5 = strchr(s1, 'y');
 s6 = strrchr(s2, 'n');
 printf ("\n%s", s3);
 printf ("\n%s", s4);
 printf ("\n%s", s5);
 printf ("\n%s", s6);
}
```

21. What would be printed from the following program block?

```
{
 char  s1[50] = "uabefgnpanm";
 char* s2 =  "ab";
```

continued

```
 char* s3  =   "pan";
 char* s4  =   "bef";
 char* s5  =  "panam";
 char* s6;
 char* s7;
 char* s8;
 char* s9;
 s6 = strstr (s1, s2);
 s7 = strstr (s1, s3);
 s8 = strstr (s1, s4);
 s9 = strstr (s1, s5);
 printf ("\n%s",  s6);
 printf ("\n%s",  s7);
 printf ("\n%s",  s8);
 printf ("\n%s",  s9);
}
```

22. What would be printed from the following program block?

```
{
 char* s1 = "abefgnpanm";
 char* s2 = "ab";
 char* s3 = "pan";
 char* s4 = "bef";
 char* s5 = "panam";
 int d1;
 int d2;
 int d3;
 int d4;
 d1 = strspn (s1, s2);
 d2 = strspn (s1, s3);
 d3 = strcspn (s1, s4);
 d4 = strcspn (s1, s5);
 printf ("\n%d",  d1);
 printf ("\n%d",  d2);
 printf ("\n%d",  d3);
 printf ("\n%d",  d4);
}
```

23. What would be printed from the following program block?

```
{
 char* w  = "BOOBOO";
 printf ("%s\n", "DOO");
 printf ("%s\n", "DICK" + 2);
 printf ("%s\n", "DOOBOO" +3);
 printf ("%c\n", w[4]);
 printf ("%s\n", w+4);
```

continued

```
    w++;
    w++;
    printf ("%s\n",w);
    printf ("%c\n", *(w+1));
   }
```

24. What would be printed from the following program block?

```
{
 char* a[5]  =
            {"GOOD", "BAD", "UGLY", "WICKED", "NICE"};
 printf ("%s\n",   a[0]);
 printf ("%s\n", *(a + 2));
 printf ("%c\n", *(a[2] + 2));
 printf ("%s\n",   a[3]);
 printf ("%s\n",   a[2]);
 printf ("%s\n",   a[4]);
 printf ("%c\n", *(a[3] +2));
 printf ("%c\n", *(*(a+4)+3));
}
```

25. What would be printed from the following program block?

```
{
 char  c[] = "programming";
 char* p;
 int   i;
 for (p = &c[5]; p >= &c [0];  p--)
      printf("%c", *p);
 printf ("\n");
 for (p = c+5, i=0; p >= c; p--,i++)
      printf("%c", *(p - i));
}
```

Problems

26. Write a function that accepts a string (a pointer to a character) and deletes the last character by moving the null character one position to the left.

27. Write a function that accepts a string (a pointer to a character) and deletes the first character.

28. Write a function that accepts a string (a pointer to a character) and deletes all the trailing spaces at the end of the string.

29. Write a function that accepts a string (a pointer to a character) and deletes all the leading spaces.

30. Write a function that returns the number of times the character is found in a string. The function has two parameters. The first parameter is a pointer to a string. The second parameter is the character to be counted.

31. Write a function that inserts a string into another string at a specified position. It returns a positive number if it is successful or zero if it has any problems, such as an insertion location greater than the length of the receiving string. The first parameter is the receiving string, the second parameter is the string to be inserted, and the third parameter is the insertion (index) position in the first string.

32. Write a function that, given a string, a width, and an empty string for output, centers the string in the output area. The function is to return 1 if the formatting is successful and 0 if any errors, such as string length greater than width, are found.

33. Write a function called `newStrCpy` that does the same job as *strcpy*. The declaration for your function is to be the same as the library function.

34. Write a function called `newStrCat` that does the same job as *strcat*. The declaration for your function is to be the same as the library function.

35. Write a function called `newStrCmp` that does the same job as *strcmp*. The declaration for your function is to be the same as the library function.

36. A string is a palindrome if it can be read forward and backward with the same meaning. Capitalization and spacing are ignored. For example, *anna* and *go dog* are palindromes. Write a function that accepts a string and returns true if the string is a palindrome and false if it is not. Test your function with the following two palindromes and at least one case that is not a palindrome:

```
Madam, I'm Adam
Able was I ere I saw Elba
```

37. Today's spelling checkers do much more than simply test for correctly spelled words. They also verify common punctuation. For example, a period must be followed by only one space. Write a program that reads a text file and removes any extra spaces after a period, comma, semicolon, or colon. Write the corrected text to a new file.

    Your program is to read the data using *fgets* and parse the strings using *strtok*. Print your test data before and after they are run through your program.

Projects

38. Write a C program that converts a string representing a number in Roman numeral form to decimal form. The symbols used in the Roman numeral system and their equivalents are given below.

I	1
V	5
X	10
L	50
C	100
D	500
M	1000

For example, the following are Roman numbers: XII (12) CII (102); XL (40). The rules for converting a Roman number to a decimal number are as follows:

a. Set the value of the decimal number to zero.

b. Scan the string containing the Roman character from left to right. If the character is not one of the symbols in the numeral symbol set, the program must print an error message and terminate. Otherwise, continue with the following steps. (Note that there is no equivalent to zero in the Roman numerals.)

□ If the next character is a null character (if the current character is the last character), add the value of the current character to the decimal value.

□ If the value of the current character is greater than or equal to the value of the next character, add the value of the current character to the decimal value.

□ If the value of the current character is less than the next character, subtract the value of the current character from the decimal value.

Solve this project using parallel arrays. Do not solve it using a *switch* statement.

39. Rework Project 38 to convert a decimal number to a Roman numeral.

40. Write a program that simulates the search-and-replace operation in a text editor. The program is to have only three function calls in *main*. The first function prompts the user to type a string of less than 80 characters. It then prompts the user to type the search substring of 10 or fewer characters. Finally, it prompts the user to type the replace substring of 10 or fewer characters.

The second call is the search-and-replace function, which replaces all occurrences of the search substring with the replace substring and creates a new string. If no occurrences are found, it returns the original string. Theoretically, the new string could be 800 characters long

(80 identical characters replaced by 10 characters each). Your function must be able to handle overflow by using the *realloc* function to extend the new string when necessary. (Start with an output string of 80 characters and extend by 80 as necessary.) The search-and-replace function returns the address of the new string.

After the search-and-replace function returns, a print function prints the resulting string as a series of 80 character lines. It performs word-wrap. That is, a line can end only at a space. If there is no space in 80 characters, then print 79 characters and a hyphen, and continue on the next line.

Write each called function using good structured programming techniques. It is expected to call subfunctions as necessary.

Run the program at least three times:

a. First, run it with no substitutions in the original input.
b. Second, run it with two or more substitutions.
c. Finally, run it with substitutions that cause the output to be at least three lines, one of which requires a hyphen.

41. Write a program that "speaks" pig-latin. Words in pig-latin are taken from English. To form a word in pig-latin, the first letter of the English word beginning with a consonant is removed and added at the end of the word, adding the letters *ay* after the moved consonant. Words that begin with a vowel are simply appended with *ay*. Thus, in pig-latin, *pig-latin* is *igpay-atinlay*.

Your program will read a sentence at a time using *fgets*. It is to parse the words into strings. As words are parsed, they are to be converted to pig-latin and printed.

42. Write a program that provides antonyms to common words. An antonym is a word with the opposite meaning. For example, the antonym of *happy* is *sad*.

The data structure is to be patterned after the Morse code problem in Section 11.7. Given a word, the program is to look for the word on both sides of the structure and, if found, report the antonym. If the word is not found, the program is to display an appropriate message.

The program is to use an interactive user interface. The words are to be read from a dictionary file. Use your dictionary to provide at least 30 antonyms, including at least one from each letter in the alphabet.

Test your program by finding the antonyms for the first word, last word, and a word somewhere in the middle on both sides of the structure. Include in your test data at least three words that are not found, one less than the first word on the left side, one in the middle somewhere, and one greater than the last word on the right side.

43. The Morse code program in this chapter is a public example of a cryptographic system. We can apply the same techniques to encoding and decoding any message. For example, we can substitute the letter Z for the letter *A*, the letter *Y* for the letter *B*, and so forth, to create the following simple encoded message:

```
NZWZN, R ZN ZWZN
MADAM, I AM ADAM
```

Write a program that encodes and decodes messages using any user-supplied code. To make it more difficult to read, include spaces and the common punctuation characters in the code. The code is to be read from a text file to build the encode/decode array. The user is then given a menu of choices to encode, decode, or enter a new code from the keyboard.

Test your program with the following code and message and with the complete alphabet entered in its encoded sequence so that it prints out in alphabetical order:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ .,?!;
?Q.W,EMRNTBXYUV!ICO PZA;SDLFKGJH
```

Message

```
WNWLSVPLM, LN A
```

44. Write a program that parses a text file into words and counts the number of occurrences of each word. Allow for up to 100 different words. After the list has been built, sort and print it.

45. Modify the program you wrote in Project 44 to eliminate common words such as *the*, *a*, *an*, *and*, *for*, and *of*. You may add other common words to the list.

Chapter 12 Enumerated, Structure, and Union Types

We have already discussed three of the six derived types: functions in Chapter 4, arrays in Chapter 8, and pointers in Chapter 9. In this chapter, we discuss the three remaining derived types: enumerated, structure, and union. We also discuss a very useful construct, the type definition. The derived types are shown in Figure 12-1.

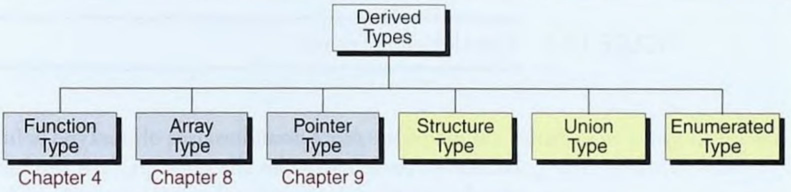


FIGURE 12-1 Derived Types

Objectives

❑ To introduce the structure, union, and enumerated types

❑ To use the type definition statement in programs

❑ To use enumerated types, including anonymous types, in programs

❑ To create and use structures in programs

❑ To be able to use unions in programs

❑ To understand the software engineering concept of coupling and to be able to evaluate coupling between functions.

12.1 The Type Definition (*typedef*)

Before discussing these derived types, let's discuss a C declaration that applies to all of them—the type definition.

A type definition, *typedef*, gives a name to a data type by creating a new type that can then be used anywhere a type is permitted. The format for the type definition is shown in Figure 12-2.

We can use the type definition with any type. For example, we can redefine **int** to `INTEGER` with the statement shown below, although we woul never recommend this.

```
typedef int INTEGER;
```

Note that the **typedef** identifier is traditionally coded in uppercase. Thi alerts the reader that there is something unusual about the type. We saw thi previously with defined constants in Chapter 2. Simple style standards suc as this make it much easier to follow a program.

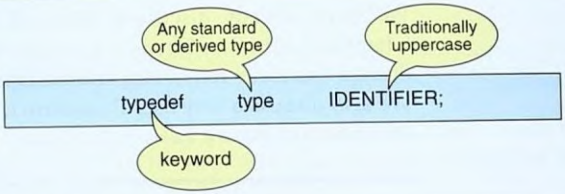


FIGURE 12-2 Type-definition Format

One of the more common uses of the type definition is with complex d larations. To demonstrate the concept, let's see how we can declare a rat simple construct, an array of pointers to strings (a ragged array). Withou type definition, we must define the array as shown below.

```
char* stringPtrAry[20];
```

We can simplify the declaration by using a type definition to creat string type and then defining the array using the new type as shown belov

```
typedef char* STRING;
STRING  stringPtrAry[20];
```

12.2 Enumerated Types

The **enumerated type** is a user-defined type based on the standard int type. In an enumerated type, each integer value is given an identifier c an **enumeration constant**. We can thus use the enumerated constants as symbolic names, which makes our programs much more readable.

Recall from Chapter 2 that a type is a set of values and a set of operations that can be applied on those values. Each type also has an identifier or name. For example, the standard type *int* has an identifier (*int*), a set of values (-∞. . . +∞), and a set of operations (such as add and multiply). While the system defines the names, values, and operations for standard types, we must define them for types we create.[1]

Declaring an Enumerated Type

To declare an enumerated type, we must declare its identifier and its values. Because it is derived from the integer type, its operations are the same as for integers. The syntax for declaring an enumerated type is:

```
enum typeName {identifier list};
```

The keyword, **enum**, is followed by an identifier and a set of enumeration constants enclosed in a set of braces. The statement is terminated with a semicolon. The enumeration identifiers are also known as an enumeration list.

Each enumeration identifier is assigned an integer value. If we do not explicitly assign the values, the compiler assigns the first identifier the value 0, the second identifier the value 1, the third identifier the value 2, and so on until all of the identifiers have a value. For example, consider an enumerated type for colors as defined in the next statement. Note that for enumeration identifiers, we use uppercase alphabetic characters.

```
enum color {RED, BLUE, GREEN, WHITE};
```

The color type has four and only four possible values. The range of the values is `0 .. 3`, with the identifier red representing the value 0, blue the value 1, green the value 2, and white the value 3.

Once we have declared an enumerated type, we can create variables from it just as we can create variables from the standard types. In fact, C allows the enumerated constants, or variables that hold enumerated constants, to be used anywhere that integers can be used. The following example defines three variables for our color type.

```
enum color productColor;
enum color skyColor;
enum color flagColor;
```

[1] When the defined type directly translates into a standard type, as with the enumerated type, the standard types may be automatically defined.

Operations on Enumerated Types

In this section we discuss storing values in enumerated types and manipulating them.

Assigning Values to Enumerated Types

After an enumerated variable has been declared, we can store values in it. Remember, however, that an enumerated variable can hold only declared values for the type. The following example defines a color variable and uses it in several statements.

```
enum color x;
enum color y;
enum color z;

x = BLUE;
y = WHITE;
z = PURPLE;                          // Error. There is no purple.
```

Similarly, once a variable has been defined and assigned a value, we can store its value in another variable of the same type. Given the previous example, the following statements are valid.

```
x = y;
z = y;
```

Comparing Enumerated Types

Enumerated types can be compared using the comparison operators. For example, given two color variables, we can compare them for equal, greater than, or less than. We can also compare them to enumeration identifiers. These comparisons are shown in the following example.

```
if (color1 == color2)
    ...
if (color1 == BLUE)
    ...
```

Another natural use of enumerated types is with the *switch* statement. Because enumerated types are derived from integer types, they may be used in the *case* expressions. For example, consider an enumerated type for the months of the year.

```
enum months
      {JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC};

enum months dateMonth;
```

Given the months type, we can use it in a switch statement as shown in the next example.

```
switch (dateMonth)
   {
    case JAN: ...
              break;
    case FEB: ...
              break;
    ...
   } // switch
```

Other Operations on Enumerated Types

In general, any operation defined for integers can be used with enumerated types. We can add, subtract, multiply, and divide enumerated types. We can also pass them to standard and application functions.

Be aware, however, that the result may not be in the defined set, which may cause the program to fail. C does not ensure that the values stored in an enumerated variable are valid.

Enumeration Type Conversion

Enumerated types can be implicitly and explicitly cast. The compiler implicitly casts an enumerated type to an integer as required. However, when we implicitly cast an integer to an enumerated type we get either a warning or an error depending on the compiler. The following example demonstrates implicit casts.

```
int        x;
enum color y;
x = BLUE;                          // Valid. x contains 1
y = 2;                             // Compiler warning
```

We can explicitly cast an integer to an enumerated type without problems. To assign y the value blue in the previous example, we could use the following code.

```
enum color y;
y = (enum color)2;         // Valid. y contains blue
```

Initializing Enumerated Constants

While the compiler automatically assigns values to enumerated types starting with 0, we can override it and assign our own values. For example, to set up an enumerated type for the months of the year, we could use the following declaration.

```
enum months
      {JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC};

enum months dateMonth;
```

While this declaration works, it could be confusing because JAN is assigned the value 0, FEB the value 1, and so forth until DEC, which is 11. To make JAN start with 1, we could use the following declaration.

```
enum months {JAN =  1, FEB, MAR, APR, MAY, JUN,
                       JUL, AUG, SEP, OCT, NOV, DEC};
```

Note that we don't have to assign initializers to every value. If we omit the initializers, the compiler assigns the next value by adding 1. To initialize the months, therefore, we simply assign the value for JAN. The rest will be automatically assigned by the compiler.

C also allows us to assign duplicate values to identifiers. For example, we could assign similar colors identical values as shown in the next example.

```
enum mycolor {RED,    ROSE = 0, CRIMSON = 0, SCARLET = 0,
              BLUE,   AQUA = 1, NAVY    = 1,
              GREEN, JADE = 2, WHITE};
```

To emphasize the point, even though RED in Color (see page 747) has the same value as RED in myColor, the two types are different and cannot be used together unless one of them is cast.

Anonymous Enumeration: Constants

If we create an enumerated type without a name, it is an anonymous enumerated type. Because the identifiers in enumerated types are constants, enumerated types are a convenient way to declare constants. For example, to assign names to common punctuation characters, we would use the following code.

```
enum {space = ' ', comma-',', colon = ':', ...};
```

As another example, to declare the constants ON and OFF, we could use the following code. Anonymous enumeration cannot be used to declare a variable.

```
enum {OFF, ON};
```

The identifier OFF is a constant with a value of 0; ON is a value of 1. As an aside, we coded OFF first because we wanted it to have a connotation of false. Similarly, ON has a connotation of true.

If you have cable TV, you know that the cable supplier is always changing the channels. To help us keep track of our favorite channels, therefore, we wrote a program that prints out the cable television channels for our local cable provider. Since the channels are changing continually, using their names in an enumerated type makes it easy to change them and to follow the code. The implementation is seen in Program 12-1.

PROGRAM 12-1 Print Cable TV Stations

```
/* Print selected TV stations for our cable TV system.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   enum TV {fox =  2, nbc  =  4, cbs  =  5,
            abc = 11, hbo  = 15, show = 17,
            max = 31, espn = 39, cnn  = 51};

// Statements
   printf("Here are my favorite cable stations:\n");
   printf("  ABC: \t%2d\n", abc);
   printf("  CBS: \t%2d\n", cbs);
   printf("  CNN: \t%2d\n", cnn);
   printf("  ESPN:\t%2d\n", espn);
   printf("  Fox: \t%2d\n", fox);
   printf("  HBO: \t%2d\n", hbo);
   printf("  Max: \t%2d\n", max);
   printf("  NBC: \t%2d\n", nbc);
   printf("  Show:\t%2d\n", show);
   printf("End of my favorite stations. \n");
   return 0;
}  // main
```

```
Results:
Here are my favorite cable stations:
  ABC:   11
  CBS:    5
  CNN:   51
  ESPN:  39
  Fox:    2
  HBO:   15
  Max:   31
  NBC:    4
  Show:  17
End of my favorite stations.
```

Input/Output Operations

Because enumerated types are derived types, they cannot be read and written using the formatted input/output functions. When we read an enumerated type, we must read it as an integer. When we write it, it displays as an integer. When necessary, C implicitly casts variables as described in "Enumeration Type Conversion" earlier in the chapter.

The following example shows how we would read and write two months.

```
enum months {JAN =  1, FEB, MAR, APR, MAY, JUN,
                       JUL, AUG, SEP, OCT, NOV, DEC};
enum months month1;
enum months month2;

scanf ("%d %d", &month1, &month2);      // Input 1 - 12
printf("%d %d", month1, month2);        // Prints 1 - 12
```

Don't be confused about strings and enumerated types. "Jan" is a string made of three characters; JAN as defined in the previous code example, is an enumerated type (identifier) which has the integer value 1.

12.3 Structure

Today's modern applications require complex data structures to support them. For example, the structures needed to support the graphical user interface found in computer workstation windows obviously are very complex. So complex, in fact, that they could not be built using the relatively primitive data types we have seen so far. A much more powerful capability is needed. This is the role of the structure.

A **structure** is a collection of related elements, possibly of different types, having a single name. For example, consider the file structure you have been using ever since you wrote your first program that used files. The file table is a type-defined structure, FILE.

Each element in a structure is called a field. A field is the smallest element of named data that has meaning. It has many of the characteristics of the variables you have been using in your programs. It has a type, and it exists in memory. It can be assigned values, which in turn can be accessed for selection or manipulation. A field differs from a variable primarily in that it is part of a structure.

We have studied another derived data type that can hold multiple pieces of data, the array. The difference between an array and a structure is that all elements in an array must be of the same type, while the elements in a structure can be of the *same or different types*.

Figure 12-3 contains two examples of structures. The first example, `fraction`, has two fields, both of which are integers. The second example, `student`, has three fields, made up of two different types.

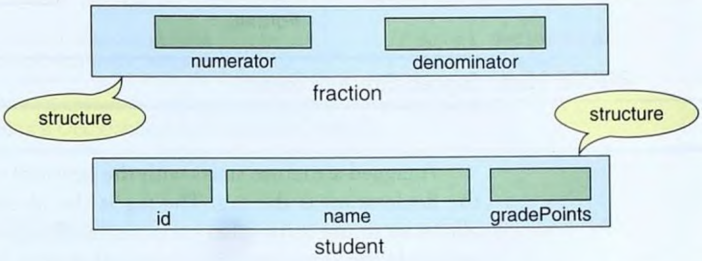


FIGURE 12-3 Structure Examples

Another way to look at a structure is as a pattern or outline that can be applied to data to extract individual parts. It allows us to refer to a collection of data using a single name and, at the same time, to refer to the individual components through their names. By collecting all the attributes of an object in one structure, we simplify our programs and make them more readable.

One design caution, however. The data in a structure should all be related to one object. In Figure 12-3, the integers in the fraction both belong to the same fraction, and the data in the second example all relate to one student. Do not combine unrelated data for programming expediency. That would not be good structured programming.

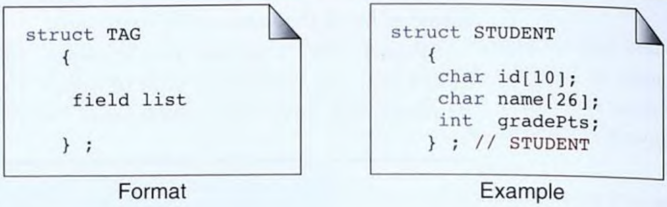
Elements in a structure can be of the same or different types. However, all elements in the structure should be logically related.

Structure Type Declaration

Like all data types, structures must be declared and defined. C has two ways to declare a structure: tagged structures and type-defined structures.

Tagged Structure

The first way to define a structure is to use a **tagged structure**. A tagged structure can be used to define variables, parameters, and return types. The format is shown in Figure 12-4.



```
struct TAG
   {

    field list

   } ;
```

Format

```
struct STUDENT
   {
    char id[10];
    char name[26];
    int  gradePts;
   } ; // STUDENT
```

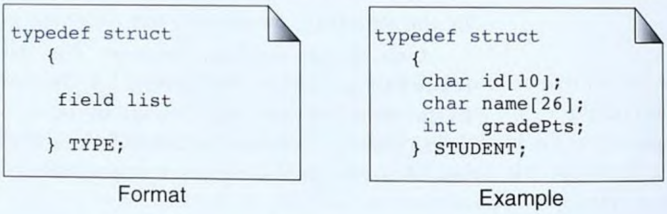
Example

FIGURE 12-4 Tagged Structure Format

A tagged structure starts with the keyword **struct.** The second element in the declaration is the **tag.** The tag is the identifier for the structure, and it allows us to use it for other purposes, such as variables and parameters. If we conclude the structure with a semicolon after the closing brace, no variables are defined. In this case, the structure is simply a type template with no associated storage.

Type Declaration with *typedef*

The more powerful way to declare a structure is to use a type definition, *typedef*. The *typedef* format is shown in Figure 12-5.



```
typedef struct
   {

    field list

   } TYPE;
```

Format

```
typedef struct
   {
    char id[10];
    char name[26];
    int  gradePts;
   } STUDENT;
```

Example

FIGURE 12-5 Structure Declaration with *typedef*

The type-defined structure differs from the tagged declaration in two ways. First, the keyword, **typedef,** is added to the beginning of the definition. Second, an identifier is required at the end of the block; the identifier is the type definition name.

Variable Declaration

After a structure has been declared, we can declare variables using it. Generally, we declare the type in the global area of a program to make it visible to all functions. The variables, on the other hand, are usually declared in the functions, either in the header or in the local declarations section. Figure 12-6 demonstrates the declaration of a structure type and variables that use it.

```
// Global Type Declarations
struct STUDENT
   {
    char id[10];
    char name[26];
    int  gradePts;
   } ;

// Local Declarations
struct STUDENT aStudent;
```

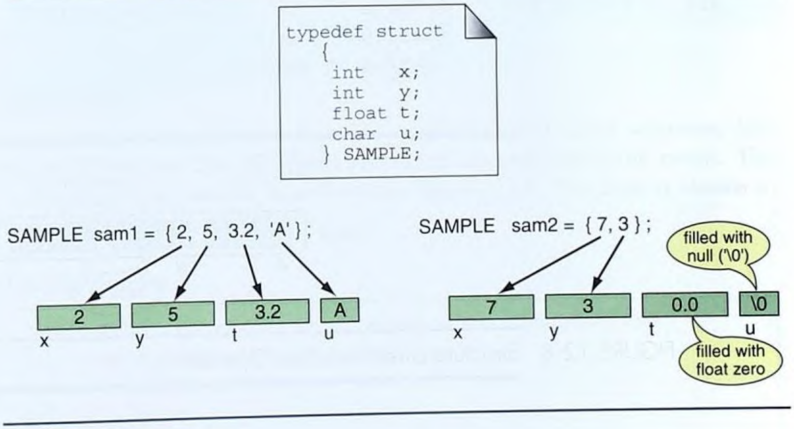
```
// Global Type Declarations
typedef struct
   {
    char id[10];
    char name[26];
    int  gradePts;
   } STUDENT;

// Local Declarations
STUDENT aStudent;
```

FIGURE 12-6 Structure Declaration Format and Example

Initialization

We can initialize a structure. The rules for structure initialization are similar to the rules for array initialization. The initializers are enclosed in braces and separated by commas. They must match their corresponding types in the structure definition. Finally, when we use a nested structure (see "Nested Structures" later in this section), the nested initializers must be enclosed in their own set of braces.

Figure 12-7 shows two examples of structure initialization. The first example shows an initializer for each field. Note how they are mapped to the structure in sequence. The second example demonstrates what happens when not all fields are initialized. As we saw with arrays, when one or more initializers are missing, the structure elements will be assigned null values, zero for integers and floating-point numbers, and null ('\0') for characters and strings.



```
typedef struct
   {
    int   x;
    int   y;
    float t;
    char  u;
   } SAMPLE;
```

SAMPLE sam1 = { 2, 5, 3.2, 'A' };

SAMPLE sam2 = { 7, 3 };

FIGURE 12-7 Initializing Structures

Accessing Structures

Now that we have discussed how to declare and initialize structures, it's time to see how we can use them in our programs. We will first describe how to access individual components of a structure and then examine the assignment of whole structures. After looking at how pointers are used with structures, we conclude with a program that uses structures.

Referencing Individual Fields

Each field in a structure can be accessed and manipulated using expressions and operators. Anything we can do with an individual variable can be done with a structure field. The only problem is to identify the individual fields we are interested in.

To refer to a field in a structure we need to refer to both the structure and the field. C uses a hierarchical naming convention that first uses the structure variable-identifier and then the field identifier. The structure variable-identifier is separated from the field identifier by a dot. The dot is the direct selection operator, which is a postfix operator at precedence 16 in the precedence table.

Using the structure student in "Type Declaration with typedef," we would refer to the individual components as shown below.

```
aStudent.id
aStudent.name
aStudent.gradePoints
```

Figure 12-8 contains another example using the structure SAMPLE, defined in Figure 12-7. With this structure, we can use a selection expression to evaluate the sam2 field, u, and if it is an A, add the two integer fields and store the result in the first. This code is shown below.

```
if (sam2.u == 'A')
    sam2.x += sam2.y;
```

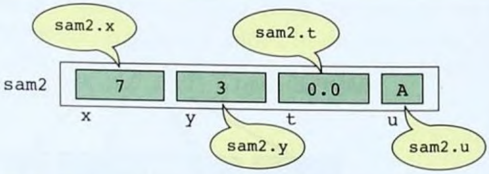


FIGURE 12-8 Structure Direct Selection Operator

We can also read data into and write data from structure members just as we can from individual variables. For example, the value for the fields of the sample structure can be read from the keyboard and placed in sam1 using the following **scanf** statement. Note that the address operator is at the beginning of the variable structure identifier.

```
scanf("%d %d %f %c",
      &sam1.x, &sam1.y, &sam1.t, &sam1.u);
```

Precedence of Direct Selection Operator

As we saw in the previous section, the direct-selection operator (.) is a postfix expression that has a priority of 16. Let's look at several examples to see how it works.

First, let's look at examples in which all operators have a priority of 16. In this case, the expressions are evaluated using left to right associativity as shown in the following examples.

```
typedef struct
   {
    int total;
    int ary[10];
   } DATA;
DATA sales;
sales.total++;          // interpreted as (sales.total)++
fun(sales.total);       // sales.total passed to fun
fun(sales.ary[3]);      // (sales.ary)[3] passed to fun
```

Now let's examine cases in which the direct selection operator is used with an operator of lesser precedence. In these cases, the direct selection is applied first.

```
++sales.total;          // interpreted as ++(sales.total)
fun(&sales.total);      // &(sales.total) passed to fun
```

EXAMPLE 12-1 Multiply Fractions

As an example of a function that uses structures and direct selection, let's look at a program that multiplies two fractions and prints the result. The structure for this example was shown in Figure 12-3. The code is shown in Program 12-2.

PROGRAM 12-2 Multiply Fractions

```
/* This program uses structures to simulate the
   multiplication of fractions.
      Written by:
      Date:
*/
```

continued

PROGRAM 12-2 Multiply Fractions *(continued)*

```c
#include <stdio.h>

// Global Declarations
   typedef  struct
      {
       int numerator;
       int denominator;
      } FRACTION;

int main (void)
{
// Local Declarations
   FRACTION  fr1;
   FRACTION  fr2;
   FRACTION  res;

// Statements
   printf("Key first fraction in the form of x/y: ");
   scanf ("%d /%d", &fr1.numerator, &fr1.denominator);
   printf("Key second fraction in the form of x/y: ");
   scanf ("%d /%d", &fr2.numerator, &fr2.denominator);

   res.numerator   = fr1.numerator   * fr2.numerator;
   res.denominator = fr1.denominator * fr2.denominator;

   printf("\nThe result of %d/%d * %d/%d is %d/%d",
          fr1.numerator, fr1.denominator,
          fr2.numerator, fr2.denominator,
          res.numerator, res.denominator);
   return 0;
} // main
```

```
Results:
Key first fraction in the form of x/y:     2/6
Key second fraction in the form of x/y:    7/4

The result of 2/6 * 7/4 is 14/24
```

Program 12-2 Analysis Although this is a very simple program, several points are of interest.

1. Note that we have coded the **typedef** in the global area before *main*. This is the first time that we have included statements here other than preprocessor directives. Even though this program has no subfunction, it is customary to put

the typed definition statements there so that they are in scope for the entire compilation unit.

2. Note that the name of the typed definition is all UPPERCASE. This is another C style tradition. It warns the reader that there is something special about this type, in this case that it is a typed definition.
3. Now examine the **scanf** statements. Since we need a pointer for **scanf**, we must use the address operator as well as the direct selection operator. As you can see, we have not used the parentheses around the field name. The reason is that the direct selection operator (`.`) has a higher priority (16) than the address operator (15). In other words, the expression

```
&fr1.numerator
```

is interpreted by the compiler as

```
&(fr.numerator)
```

which is exactly what we need.

Operations on Structures

The structure is an entity that can be treated as a whole. However, only one operation, assignment is allowed on the structure itself. That is, a structure can only be copied to another structure of the same type using the assignment operator.

Rather than assign individual members when we want to copy one structure to another, as we did earlier, we can simply assign one to the other. Figure 12-9 copies `sam1` to `sam2`

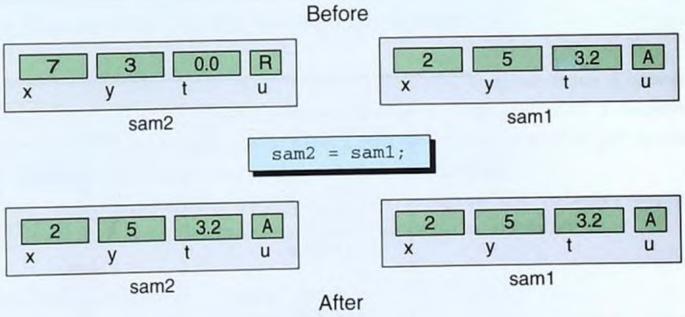
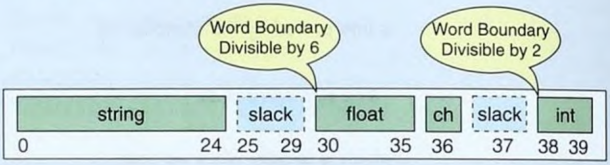


FIGURE 12-9 Copying a Structure

Slack Bytes

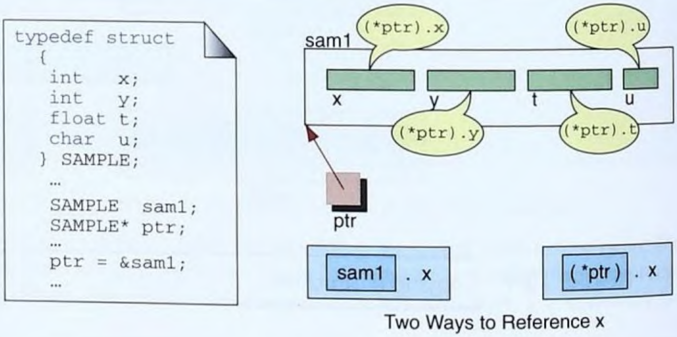
It is interesting to examine why we cannot compare two structures. Sometimes hardware requires that certain data, such as integers and floating-point numbers, be aligned on a word boundary in memory. When we group data in a structure, the arrangement of the data may require that **slack bytes** be inserted to maintain these boundary requirements. For example, consider the structure shown below. In this structure we assume that a floating-point number is stored in a word that requires 6 bytes and must be on an address evenly divisible by 6, such as 24 or 30. We also assume that integers are stored in 2-byte words that require an address evenly divisible by 2. The 25-byte string at the beginning of the structure forces 5 slack bytes between the string and the float. Then the character after the float forces a slack byte to align the integer at the end of the structure.



Since these extra bytes are beyond the control of the program, we cannot guarantee what their values will be. Therefore, if we were to compare two structures and their first components were equal, the inserted slack bytes could cause an erroneous compare result, either high or low, if they weren't equal. C prevents this problem by not allowing selection statements with structures. Of course, if we really need to compare structures, we can simply write a structure compare function of our own that would compare the individual fields in the structure.

Pointer To Structures

Structures, like other types, can also be accessed through pointers. In fact, this is one of the most common methods used to reference structures. For example, let's use our `SAMPLE` structure with pointers (Figure 12-10).



```
typedef struct
  {
   int   x;
   int   y;
   float t;
   char  u;
  } SAMPLE;
   …
   SAMPLE  sam1;
   SAMPLE* ptr;
   …
   ptr = &sam1;
   …
```

FIGURE 12-10 Pointers to Structures

The first thing we need to do is define a pointer for the structure as shown below.

```
SAMPLE* ptr;
```

We now assign the address of `sam1` to the pointer using the address operator (`&`) as we would with any other pointer.

```
ptr = &sam1;
```

Now we can access the structure itself and all the members using the pointer, `ptr`. The structure itself can be accessed like any object using the indirection operator (`*`).

```
*ptr                          // Refers to whole structure
```

Since the pointer contains the address of the beginning of the structure, we no longer need to use the structure name with the direct selection operator. The pointer takes its place. The reference to each of the `sample` members is shown below and in Figure 12-10.

```
(*ptr).x    (*ptr).y    (*ptr).t    (*ptr).u
```

Note the parentheses in the above expressions. *They are absolutely necessary,* and to omit them is a very common mistake. The reason they are needed is that the precedence priority of the direct selection operator (`.`) is higher than the indirection operator (`*`). If we forget to put the parentheses, C applies the dot operator first and the asterisk operator next. In other words,

`*ptr.x` *is interpreted as* `*(ptr.x)`

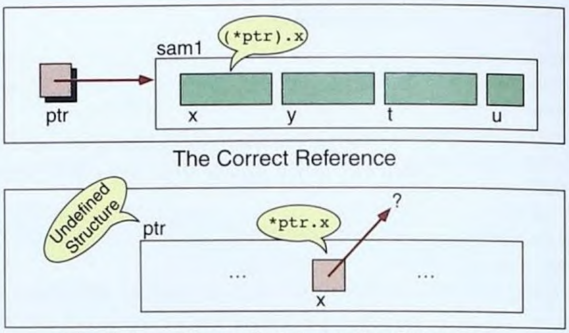
which is wrong. The expression `*(ptr.x)` means that we have a completely different (and undefined) structure called `ptr` that contains a member, `x`, which must be a pointer. Since this is not the case, you will get a compile error. Figure 12-11 shows how this error is interpreted.

The correct notation, `(*ptr).x`, first resolves the primary expression (`*ptr`) and then uses the pointer value to access the member, `x`.

Indirect Selection Operator

Fortunately, another operator— indirect selection—eliminates the problems with pointers to structures. The indirect selection operator is at the same level in the Precedence Table (see inside front cover) as the direct selection operator.

`(*pointerName).fieldName` ↔ `pointerName->fieldName`

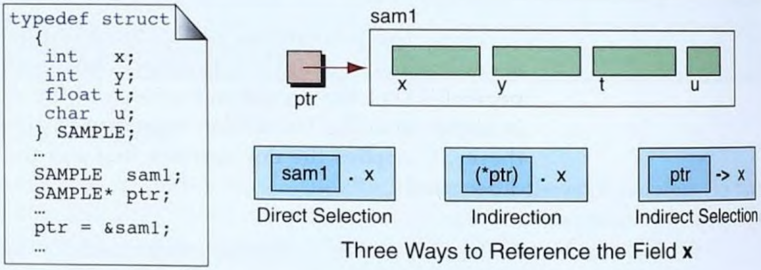


The Correct Reference

The Wrong Way to Reference the Component

FIGURE 12-11 Interpretation of Invalid Pointer Use

The token for the indirect selection operator is an arrow formed by the minus sign and the greater than symbol (->). It is placed immediately after the pointer identifier and before the member to be referenced. We use this operator to refer to the members of our previously defined structure, `sam1`, in Figure 12-12.



```
typedef struct
  {
   int   x;
   int   y;
   float t;
   char  u;
  } SAMPLE;
  ...
  SAMPLE  sam1;
  SAMPLE* ptr;
  ...
  ptr = &sam1;
  ...
```

Three Ways to Reference the Field **x**

FIGURE 12-12 Indirect Selection Operator

Example: Clock Structure

Program 12-3 simulates a digital clock that shows time. A structure is defined to represent the three components of time: hour, minute, and second. Two functions are used. The first function, called `increment`, simulates the passage of the time. The second function, called `show`, shows the time at any moment.

PROGRAM 12-3 Clock Simulation with Pointers

```
/* This program uses a structure to simulate the time.
      Written by:
```

continued

PROGRAM 12-3 Clock Simulation with Pointers *(continued)*

```
      Date:
*/
#include <stdio.h>

typedef struct
   {
    int hr;
    int min;
    int sec;
   } CLOCK;

// Function Declaration
void increment (CLOCK* clock);
void show      (CLOCK* clock);

int main (void)
{
// Local Declaration
   CLOCK clock = {14, 38, 56};

// Statements
   for(int i = 0; i < 6; ++i)
      {
       increment (&clock);
       show (&clock);
      } // for
   return 0;
}  // main

/* =================== increment ====================
   This function accepts a pointer to clock and
   increments the time by one second.
      Pre   previous clock setting
      Post  clock incremented by one second.
*/
void increment (CLOCK* clock)
{
// Statements
   (clock->sec)++;
   if (clock->sec == 60)
      {
       clock->sec = 0;
       (clock->min)++;
       if (clock->min == 60)
```

continued

PROGRAM 12-3 Clock Simulation with Pointers *(continued)*

```
          {
           clock->min = 0;
           (clock->hr)++;
           if (clock->hr == 24)
               clock->hr = 0;
          } // if 60 min
      } // if 60 sec
   return;
}  // increment

/* ===================== show =====================
   Show the current time in military form.
      Pre   clock time
      Post  clock time displayed
*/
void show (CLOCK* clock)
{
// Statements
   printf("%02d:%02d:%02d\n",
           clock->hr, clock->min, clock->sec);
   return;
}  // show
```

```
Results:
14:38:57
14:38:58
14:38:59
14:39:00
14:39:01
14:39:02
```

Complex Structures

As mentioned earlier, structures were designed to deal with complex problems. The limitations on structures are not on the structures themselves but on the imagination of the software engineers who solve the problems. Structures within structures (nested structures), arrays within structures, and arrays of structures are all common. We deal with the first two here and arrays of structures in the next section.

Nested Structures

We can have structures as members of a structure. When a structure includes another structure, it is a **nested structure**. There is no limit to the number of structures that can be nested, but we seldom go beyond three.

For example, we can have a structure called `stamp` that stores the date and the time. The date is in turn a structure that stores the month, day, and year. The time is also a structure that stores the hour, minute, and second. This structure design is shown in Figure 12-13.

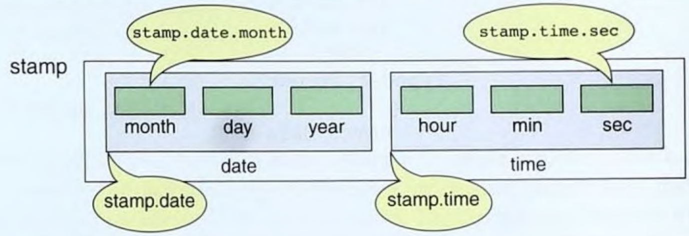


FIGURE 12-13 Nested Structure

There are two concerns with nested structures: declaring them and referencing them.

Declaring Nested Structures

Although it is possible to declare a nested structure with one declaration, it is not recommended. It is far simpler and much easier to follow the structure if each structure is declared separately and then grouped in the high-level structure.

When we declare the structures separately, the most important point we must remember is that nesting must be done from inside out—that is, from the lowest level to the most inclusive level. In other words, we must declare the innermost structure first, then the next level, working upward toward the outer, most inclusive structure.

Consider the time stamp structure seen in Figure 12-13. The inner two structures, `date` and `time`, must be declared before the outside structure, `stamp`, is declared. We show the declaration of `stamp` and a variable that uses it in the following example.

```
// Type Declaration
typedef struct
   {
    int month;
    int day;
    int year;
   } DATE;
typedef struct
   {
    int hour;
    int min;
    int sec;
   } TIME;
typedef struct
   {
    DATE  date;
    TIME  time;
   } STAMP;

// Variable Declaration
STAMP stamp;
```

It is possible to nest the same structure type more than once in a declaration. For example, consider a structure that contains start and end times for a job. Using the `STAMP` structure, we create a new declaration as shown below.

```
// Type Declaration
typedef struct
   {
    …
    STAMP startTime;
    STAMP endTime;
   } JOB;

// Variable Declaration
JOB job;
```

Regardless of how we declared the structure, using it will be the same. The major advantage of declaring each of the structures separately is that it allows much more flexibility in working with them. For example, with `DATE` declared as a separate type definition, it is possible to pass the date structure to a function without having to pass the rest of the `stamp` structure.

Referencing Nested Structures

To access a nested structure, we include each level from the highest (`stamp`) to the component being referenced. The complete set of references for `stamp` is shown below. The last two references are to `job`.

```
stamp
stamp.date
stamp.date.month
stamp.date.day
stamp.date.year
stamp.time
stamp.time.hour
stamp.time.min
stamp.time.sec

job.startTime.time.hour
job.endTime.time.hour
```

Nested Structure Initialization

Initialization follows the rules mentioned for a simple structure. Each structure must be initialized completely before proceeding to the next member. Each structure is enclosed in a set of braces. For example, to initialize stamp, first we initialize date, then time, separated by a comma. To initialize date, we provide values for month, day, and year, each separated by commas. We can then initialize the members of time. A definition and initialization for stamp are shown below.

```
STAMP stamp = {{05, 10, 1936}, {23, 45, 00}};
```

Structures Containing Arrays

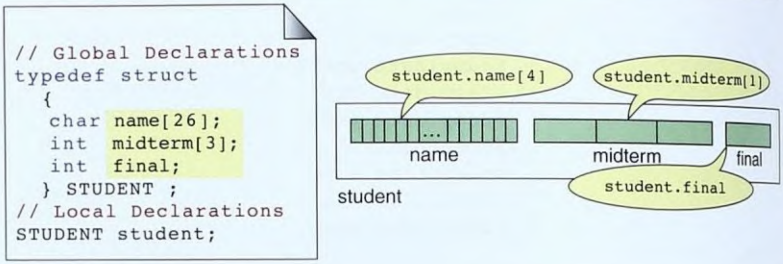
Structures can have one or more arrays as members. The arrays can be accessed either through indexing or through pointers, as long as they are properly qualified with the direct selection operator.

Defining Arrays for Structures

As with nested structures, an array may be included within the structure or may be declared separately and then included. If it is declared separately, the declaration must be complete before it can be used in the structure. For example, consider the structure that contains the student name, three midterm scores, and the final exam score, as shown in Figure 12-14.

Referencing Arrays in Structures

Regardless of how we declared the structure, each element will have the same reference. First we refer to the structure, then to the array component. When we refer to the array, we can use either index or pointer notation. Let us look at each in turn.



```
// Global Declarations
typedef struct
  {
   char name[26];
   int  midterm[3];
   int  final;
  } STUDENT ;
// Local Declarations
STUDENT student;
```

FIGURE 12-14 Arrays in Structures

The index applies only to elements within an array. Therefore, it must follow the identifier of an array. Our student example has two arrays, one of characters (a string) and the other of midterm scores. Each of these arrays can be referenced with an index as shown below.

```
student
student.name
student.name[i]
student.midterm
student.midterm[j]
student.final
```

We have already seen how to refer to fields in a structure using the indirect selection operator (–>). When one structure contains an array, we can use a pointer to refer directly to the array elements. For example, given a pointer to integer, pScores, we could refer to the scores in student as shown below.

```
pScores     = student.midterm;
totalScores = *pScores + *(pScores + 1) + *(pScores + 2);
```

Array Initialization in Structures

The initialization of a structure containing an array is simply an extension of the rules for structure initialization to include the initialization of the array. Since the array is a separate member, its values must be included in a separate set of braces. For example, the student structure can be initialized as shown below.

```
STUDENT student = {"John Marcus", {92, 80, 70}, 87};
```

Note that the name is initialized as a string and the midterm scores are simply enclosed in a set of braces.

Structure Containing Pointers

It should not be surprising that a structure can have pointers as members. In fact, we will see that pointers are very common in structures.

The use of pointers can save memory. For example, suppose that we wanted to use the alphabetic month in our stamp structure rather than an integer month. We could add an array of nine characters to each structure element, but it would be much more memory efficient to use a 4-byte pointer if we had to store a lot of these structures. Given the months of the year defined as strings, as shown below, we could then use a pointer to the correct month.

```
char jan[] = "January";
char feb[] = "February";
…
char dec[] = "December";
```

To assign the month May to the structure, we would use the following statement. Note that we are copying a pointer to a string, not the string itself.

```
stamp.date.month = may;
```

The modified structure is shown in Figure 12-15. The structure code for the date structure is shown below. It is important to remember that declaring a structure does not allocate memory for it. Even when the structure is defined, however, there is still no memory provided for month until we allocate it using *malloc* or *calloc*.

```
typedef struct
   {
    char* month;
    int  day;
    int   year;
   } DATE;
```

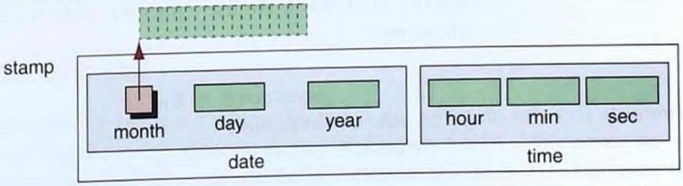


FIGURE 12-15 Pointers in Structures

Array of Structures

In many situations, we need to create an array of structures. To name just one example, we would use an array of students to work with a group of students stored in a structure. By putting the data in an array, we can quickly and easily work with the data, to calculate averages, for instance.

Let's create an array to handle the scores for up to 50 students in a class. Figure 12-16 shows how such an array might look.

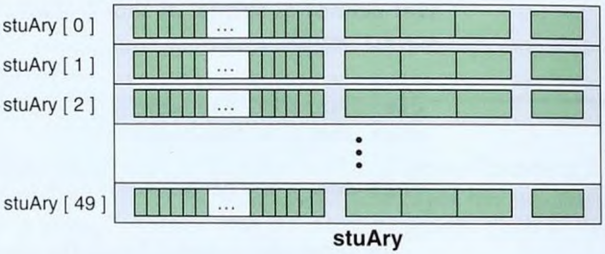


FIGURE 12-16 Array of Structures

Since a structure is a type, we can create the array just as we would create an array of integers. The code is shown below.

```
STUDENT stuAry[50];
```

Study this array carefully. Note that it is an array that contains two other arrays, name and midterm. This is not a multidimensional array. To be a multidimensional array, each level must have the same data type. In this case, each type is different: the stuAry is STUDENT, while name is character, and midterm is integer.

To access the data for one student, we refer only to the structure name with an index or a pointer as shown below.

```
              stuAry[i]          *pStu
```

For example, let's write a short segment of code to compute the average for the final exam. We use a *for* loop since we know the number of students in the array.

```
int       totScore = 0;
float     average;
STUDENT*  pStu;
STUDENT*  pLastStu;
...
```

continued

```
pLastStu  = stuAry + 49;
for (pStu = stuAry; pStu <= pLastStu; pStu++)
   totScore += pStu->final;
average = totScore / 50.0;
```

However, to access an individual element in one of the student's arrays, such as the second midterm for the fifth student, we need to use an index or pointer for each field as shown below.

```
stuAry[4].midterm[1]
```

To access students' midterms with pointers, we need one index or pointer for the array. We also need a second index or pointer for the midterms. The code to compute the average for each midterm is shown below. We use a separate array, `midTermAvrg`, to store the average for the midterms. In this example, we use indexes to access the midterms and pointers to access the students.

```
float    midTermAvrg[3];
int      sum;
STUDENT* pStu;
...
STUDENT* pLastStu;
pLastStu    = stuAry + 49;
for (int i = 0; i < 3; i++)
    {
     sum = 0;
     for (pStu = stuAry; pStu <= pLastStu; pStu++)
        sum += pStu->midterm[i];
     midTermAvrg[i] = sum / 50.0;
    } // for i
```

The Precedence Table (see inside front cover) shows that the index operator, the direct selection operator, and the indirect selection operator all have the same precedence and that the associativity is from left to right. So we do not need parentheses to total the scores for a midterm.

Insertion Sort Revisited

To demonstrate using structures in arrays, let's sort an array of students. We will use the `student` structure seen in Figure 12-14. Whenever we sort a structure, we need to define the field that controls the sort. This control field is usually called a key. In our `student` structure, the key field is the name. Program 12-4 shows the code.

PROGRAM 12-4 Sort Array of Student Structures

```
/* This program sorts an array of student structures
      Written by:
      Date:
*/
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

#define NUM_STU 5

// Global Type Declaration
   typedef struct
      {
       char  name[26];
       int   midterm[3];
       int   final;
      } STUDENT;

// Function Declarations
void insertionSort (STUDENT list[], int last);

int main (void)
{
// Local Declarations
   STUDENT  stuAry[NUM_STU] =
      {
       {"Charles, George", {85, 94, 79}, 93},
       {"Adams, Karin",    {75, 91, 89}, 89},
       {"Nguyen, Tuan",    {87, 88, 89}, 90},
       {"Oh, Bill",        {78, 96, 88}, 91},
       {"Chavez, Maria",   {83, 79, 93}, 91}
      } ; // stuAry

// Statements
   printf("Unsorted data:\n");
   for (STUDENT* pStuPtr = stuAry;
                 pStuPtr < stuAry + NUM_STU;
                 pStuPtr++)
      printf("%-26s %4d %4d %4d %4d\n",
              pStuPtr->name,
              pStuPtr->midterm[0],
              pStuPtr->midterm[1],
              pStuPtr->midterm[2],
              pStuPtr->final);
```

continued

PROGRAM 12-4 Sort Array of Student Structures *(continued)*

```
   printf("\n");

   insertionSort (stuAry, NUM_STU - 1);

   printf("Sorted data:\n");
   for (STUDENT* pStuPtr = stuAry;
                 pStuPtr < stuAry + NUM_STU;
                 pStuPtr++)
      printf("%-26s %4d %4d %4d %4d\n",
              pStuPtr->name,
              pStuPtr->midterm[0],
              pStuPtr->midterm[1],
              pStuPtr->midterm[2],
              pStuPtr->final);
   return 0;
}  // main

/* ================== insertionSort ==================
   Sort list using Insertion Sort. The list is divided
   into sorted and unsorted lists. With each pass, the
   first element in unsorted list is inserted into
   sorted list.
      Pre   list must contain at least one element
            last is index to last element in list
      Post  list has been rearranged.
*/
void insertionSort (STUDENT  list[], int last)
{
// Local Declarations
   bool     located;
   STUDENT  temp;
   STUDENT* pCurrent;
   STUDENT* pWalker;
   STUDENT* pLast;

// Statements
   for (pCurrent = list + 1, pLast = list + last;
        pCurrent <= pLast;
        pCurrent++)
     {
      located =  false;
      temp    = *pCurrent;

      for  (pWalker  = pCurrent - 1;
```

continued

PROGRAM 12-4 Sort Array of Student Structures *(continued)*

```
            pWalker >= list && !located;
           )
         if (strcmp(temp.name, pWalker->name) < 0)
            {
             *(pWalker + 1) = *pWalker;
             pWalker--;
            } // if
         else
            located = true;
      *(pWalker + 1)  = temp;
     } // for pCurrent
   return;
}  // insertionSort
// ================= End of Program =================
```

```
Results:
Unsorted data:
Charles, George                 85   94   79   93
Adams, Karin                    75   91   89   89
Nguyen, Tuan                    87   88   89   90
Oh, Bill                        78   96   88   91
Chavez, Maria                   83   79   93   91

Sorted data:
Adams, Karin                    75   91   89   89
Charles, George                 85   94   79   93
Chavez, Maria                   83   79   93   91
Nguyen, Tuan                    87   88   89   90
Oh, Bill                        78   96   88   91
```

Program 12-4 Analysis This program follows the insertion sort design we studied in Chapter 8 (Program 8-11). There are only two significant changes: First, all the array handling is done with pointers, and second, the compare logic was changed to use the *strcmp* function.

To test the sort, we wrote a small driver that uses an initialized array for its data. To verify that the sort works, we printed the data before and after the sort process.

Structures and Functions

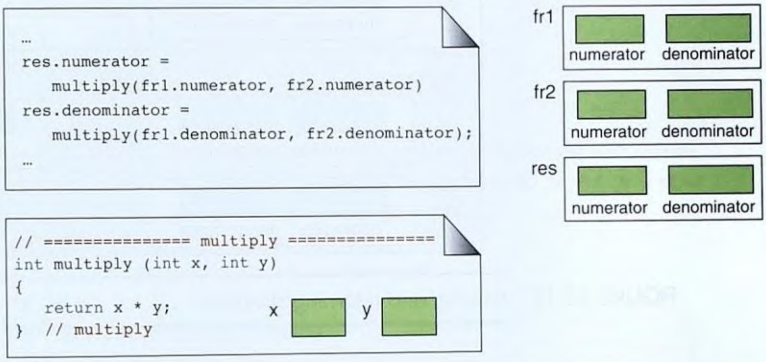
For structures to be fully useful, we must be able to pass them to functions and return them. A function can access the members of a structure in three ways:

1. Individual members can be passed to the function.
2. The whole structure can be passed and the function can access the members using pass by value.

3. The address of a structure or member can be passed, and the function can access the members through indirection and indirect selection operators.

Sending Individual Members

Sending individual members to a function is no different from what we have done since Chapter 4. To demonstrate the calling sequence, we will use a simple example to multiply two fractions. The flow is shown in Figure 12-17.



```
...
res.numerator =
   multiply(fr1.numerator, fr2.numerator)
res.denominator =
   multiply(fr1.denominator, fr2.denominator);
...
```

```
// ============== multiply ===============
int multiply (int x, int y)
{
   return x * y;
}  // multiply
```

FIGURE 12-17 Passing Structure Members to Functions

As you can see from Figure 12-17, the only difference is that we must use the direct selection operator to refer to the individual members for the actual parameters. The called program doesn't know if the two integers are simple variables or structure members.

Sending The Whole Structure

The problem with the above solution is that the multiplication logic is split between the calling and called programs. This is not considered good structured programming. A much better solution is to pass the entire structure and let `multiply` complete its job in one call.

Passing a structure is really no different from passing individual elements. Since the structure is a type, we simply specify the type in the formal parameters of the called function. Similarly, the function can return a structure. Again, all that is necessary is to specify the structure as the return type in the called function.

The same pass-by-value rules apply, however; when we pass a structure to a function, C will copy the values to the local structure just as it does for variables. This may lead to some inefficiencies when large structures are used. We address this problem in the next section.

Let us rework the multiply fractions program to pass structures. In this case, we have written the complete program. The design is shown in Figure 12-18, and the code is shown in Program 12-5.

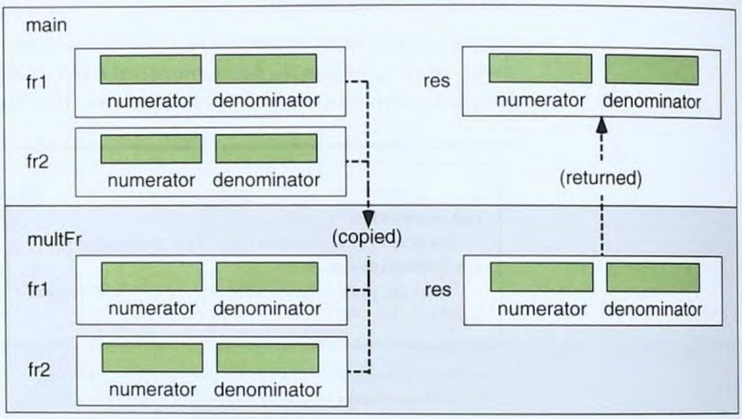


FIGURE 12-18 Passing and returning structures

PROGRAM 12-5 Passing and Returning Structures

```
/* This program uses structures to multiply fractions.
      Written by:
      Date:
*/
#include <stdio.h>

// Global Declarations
typedef  struct
    {
     int numerator;
     int denominator;
    } FRACTION;

// Function Declarations
FRACTION getFr   (void);
FRACTION multFr  (FRACTION fr1, FRACTION fr2);
void     printFr (FRACTION fr1, FRACTION fr2,
                  FRACTION result);

int main (void)
{
```

continued

PROGRAM 12-5 Passing and Returning Structures *(continued)*

```
// Local Declarations
   FRACTION fr1;
   FRACTION fr2;
   FRACTION res;

// Statements
   fr1 = getFr  ();
   fr2 = getFr  ();
   res = multFr (fr1, fr2);
   printFr (fr1, fr2, res);
   return 0;
}  // main

/* ==================== getFr ====================
   Get two integers from the keyboard, make & return
   a fraction to the main program.
      Pre   nothing
      Post  returns a fraction
*/
FRACTION getFr (void)
{
// Local Declarations
   FRACTION fr;

// Statements
   printf("Write a fraction in the form of x/y: ");
   scanf ("%d/%d", &fr.numerator, &fr.denominator);
   return fr;
}  // getFraction

/* ==================== multFr ====================
   Multiply two fractions and return the result.
      Pre   fr1 and fr2 are fractions
      Post  returns the product
*/
FRACTION multFr (FRACTION fr1, FRACTION fr2)
{
// Local Declaration
   FRACTION res;

// Statements
   res.numerator   = fr1.numerator   * fr2.numerator;
   res.denominator = fr1.denominator * fr2.denominator;
   return res;
```

continued

PROGRAM 12-5 Passing and Returning Structures *(continued)*

```
} // multFr

/* ==================== printFr ====================
   Prints the value of the fields in three fractions.
      Pre   two original fractions and the product
      Post  fractions printed
*/
void  printFr  (FRACTION fr1,  FRACTION fr2,
                FRACTION res)
{
// Statements
   printf("\nThe result of %d/%d * %d/%d is %d/%d\n",
          fr1.numerator, fr1.denominator,
          fr2.numerator, fr2.denominator,
          res.numerator, res.denominator);
   return;
}  // printFractions
// ================== End of Program =================
```

```
Results:
Write a fraction in the form of x/y: 4/3
Write a fraction in the form of x/y: 6/7

The result of 4/3 * 6/7 is 24/21
```

Program 12-5 Analysis There are five points you should study in this program.

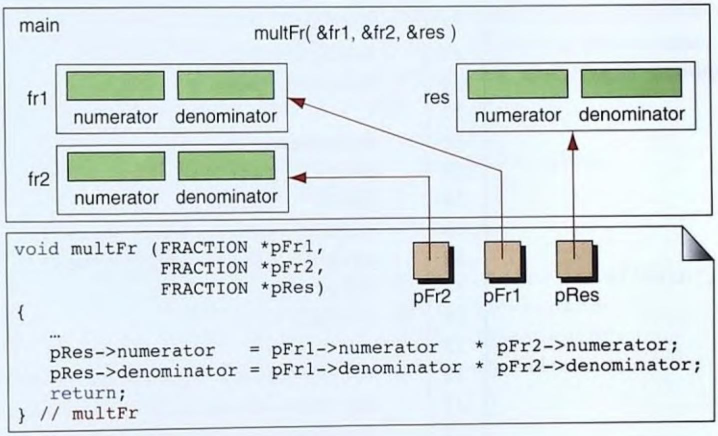
1. The fraction structure is declared in global memory before *main*. This makes it visible to all the functions.
2. In `getFr`, the data are read by using the address operator and the direct selection operator. Since the direct selection operator (`.`) has a higher priority, they can be coded without parentheses.
3. In `getFr` we pass back two values in the structure without using pointers. This is another advantage of structures: We can return more than one piece of data when we put the data in a structure.
4. The structure type is passed using the `FRACTION` type definition.
5. The references to the data in `multFr` and `printFr` must use the direct selection operator to get to the individual members.

Passing Structures Through Pointers

As mentioned, passing structures is still pass by value. For the multiply fractions program, this is the correct way to write the program. It provides the necessary data protection and data encapsulation for good structured programming while at the same time being efficient (see "Software Engineering" at the end of the chapter).

When the structure is very large, however, efficiency can suffer, especially with a heavily used function. Therefore, we often use pointers to pass structures. It is also common to pass structures through pointers when the structure is in dynamic memory. In these cases, all we have to pass is the pointer.

We modify the multiply fractions program once more to pass the structures using pointers. The memory flow is shown in Figure 12-19 and the coding in Program 12-6.



```
void multFr (FRACTION *pFr1,
             FRACTION *pFr2,
             FRACTION *pRes)
{
   ...
   pRes->numerator   = pFr1->numerator   * pFr2->numerator;
   pRes->denominator = pFr1->denominator * pFr2->denominator;
   return;
} // multFr
```

FIGURE 12-19 Passing Structures Through Pointers

PROGRAM 12-6 Passing Structures through Pointers

```
/* This program uses structures to multiply fractions.
      Written by:
      Date:
*/
#include <stdio.h>

// Global Declarations
typedef struct
        {
         int numerator;
         int denominator;
        } FRACTION;

```

continued

PROGRAM 12-6 Passing Structures through Pointers *(continued)*

```
// Function Declarations
void getFr   (FRACTION* pFr);
void multFr  (FRACTION* pFr1, FRACTION* pFr2,
              FRACTION* pRes2);
void printFr (FRACTION* pFr1, FRACTION* pFr2,
              FRACTION* pRes);

int  main (void)
{
// Local Declarations
   FRACTION fr1;
   FRACTION fr2;
   FRACTION res;

// Statements
   getFr   (&fr1);
   getFr   (&fr2);
   multFr  (&fr1, &fr2, &res);
   printFr (&fr1, &fr2, &res);
   return 0;
} // main

/* ==================== getFr ====================
   Get two integers from the keyboard, make & return a
   fraction to the main program.
      Pre   pFr is pointer to fraction structure
      Post  fraction stored at pFr.
*/
void getFr (FRACTION* pFr)
{
// Statements
   printf("Write a fraction in the form of x/y: ");
   scanf ("%d/%d", &pFr->numerator,
          &(*pFr).denominator);
   return;
} // getFr

/* ===================== multFr ====================
   Multiply two fractions and return the result.
      Pre   fr1, fr2, pRes are pointers to fractions
      Post  product stored at pRes
*/
void multFr  (FRACTION* pFr1, FRACTION* pFr2,
```

continued

PROGRAM 12-6 Passing Structures through Pointers *(continued)*

```
                 FRACTION* pRes)
{
// Statements
   pRes->numerator   =
         pFr1->numerator * pFr2->numerator;
   pRes->denominator =
         pFr1->denominator * pFr2->denominator;
   return;
}  // multFr

/* ===================== printFr =====================
   Prints the value of the fields in three fractions.
      Pre   pointers to two fractions and their product
      Post  fractions printed
*/
void printFr (FRACTION* pFr1, FRACTION* pFr2,
              FRACTION* pRes)
{
// Statements
   printf("\nThe result of %d/%d * %d/%d is %d/%d\n",
          pFr1->numerator, pFr1->denominator,
          pFr2->numerator, pFr2->denominator,
          pRes->numerator, pRes->denominator);
   return;
}  // printFr
// ================== End of Program ==================
```

Program 12-6 Analysis In this version of the program, the structure is passed and returned as a pointer. Note the syntactical notation for reading the data in the `getFr` function. We reproduce it below for your convenience.

```
scanf("%d/%d", &pFr->numerator, &(*pFr).denominator);
```

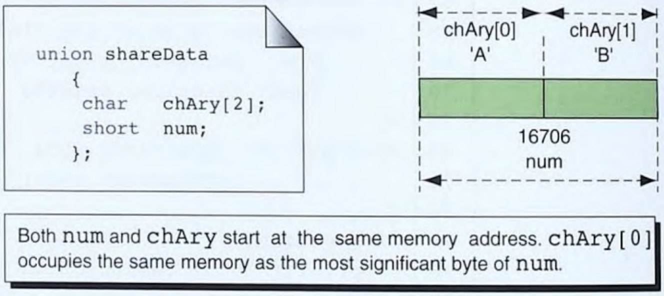
Even though we are using pointers, we still need to pass **scanf** addresses. We have used two different notations in this example. (This is not good coding style, but it demonstrates both techniques in one statement.) In both cases, since we are reading a field within the structure, we need to pass the address of the individual field, not the structure. In the first example, we use the indirect selection operator (->). Since it has a higher precedence than the address operator (&), it can be coded without parentheses.

In the second example, we use the direct selection operator (.). In this case we need parentheses because the direct selection operator has a higher precedence (16) than the indirection operator (*) and the address operator, both 15. However, since the address and direct selection operator are at the same level, we need to use the parentheses only around the pointer dereference.

12.4 Unions

The ***union*** is a construct that allows memory to be shared by different types of data. This redefinition can be as simple as redeclaring an integer as four characters or as complex as redeclaring an entire structure. For example, we know that a short integer is 2 bytes and that each byte is a character. Therefore, we could process a short integer as a number or as two characters.

The union follows the same format syntax as the structure. In fact, with the exception of the keywords *struct* and *union*, the formats are the same. Figure 12-20 shows how we declare a union that can be used as either a short integer or as two characters.



```
union shareData
   {
    char   chAry[2];
    short  num;
   };
```

Both num and chAry start at the same memory address. chAry[0] occupies the same memory as the most significant byte of num.

FIGURE 12-20 Unions

Referencing Unions

The rules for referencing a union are identical to those for structures. To reference individual fields within the *union*, we use the direct selection (dot) operator. Each reference must be fully qualified from the beginning of the structure to the element being referenced. This includes the name of the *union* itself. When a *union* is being referenced through a pointer, the selection operator (arrow) can be used. The following are both valid references to the *union* shown in Figure 12-20.

```
shareData.num
shareData.chAry[0]
```

Initializers

While C permits a *union* to be initialized, only the first type declared in the *union* can be initialized when the variable is defined. The other types can only be initialized by assigning values or reading values into the *union*. When initializing a *union*, we must enclose the values in a set of braces, even if there is only one value.

Program 12-7 demonstrates unions in a program. It uses the structure in Figure 12-20 to print a variable, first as a number and then as two characters.

PROGRAM 12-7 Demonstrate Effect of Union

```c
/* Demonstrate union of short int and two char.
     Written by:
     Date:
*/
#include <stdio.h>

// Global Declarations
   typedef union
      {
       short  num;
       char   chAry[2];
      } SH_CH2;

int main (void)
{
// Local Declarations
   SH_CH2 data;

// Statements
   data.num = 16706;

   printf("Short: %hd\n", data.num);
   printf("Ch[0]: %c\n",  data.chAry[0]);
   printf("Ch[1]: %c\n",  data.chAry[1]);

   return 0;
}  // main
```

```
Results:
Short: 16706
Ch[0]: A
Ch[1]: B
```

Program 12-7 Analysis To prove to yourself that 16706 creates the characters A and B, you will need to analyze its bit pattern. Don't be surprised, however, if you get a different number on your computer. If you do, it means that you have a "little-endian" computer (see the box "Big and Little Endian," later in this section).

Unions and Structures

In a *union*, each piece of data starts at the same memory location and occupies at least a part of the same memory. Thus, when a *union* is shared by two or more different data types, only one piece of data can be in memory at one time.

Consider, for example, an address book that contains both company and individual names. When we store a company name, the name has only one field. On the other hand, when we store an individual's name, the name has at least three parts—first name, middle initial, and last name. If we want to have only one name field in our address book, we need to use a *union* to store the name. This design is shown in Figure 12-21. Note that in the figure, un is the member name to access the *union* structure.

```
typedef struct
  {
   char  first[20];
   char  init;
   char  last[30];
  } PERSON;
typedef struct
  {
   char  type;
   union
     {
      char   company[40];
      PERSON person;
     } un;
  } NAME;
```

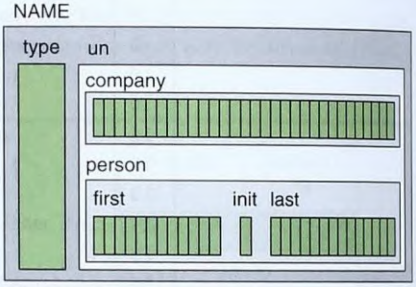


FIGURE 12-21 A Name Union

Because two different types of data can be stored in the same *union*, we need to know which type currently is stored. This is done with a type flag that is created when the data are stored. If the name is in the company format, the flag is *C*, and if it is in the person format it is a *P*.

When a *union* is defined, C reserves enough room to store the *largest* data object in the construct. In Figure 12-21, the size of the company name is 40 bytes and the size of the person name is 51 bytes. The size of the union structure is therefore 52 bytes (the size of the person name plus the type) regardless of what type of data are being stored. When a company name is stored, the last 11 bytes are present but unused.

Program 12-8 demonstrates the use of a *union* in a structure by printing each format.

PROGRAM 12-8 Demonstrate Unions in Structures

```
/* Demonstrate use of unions in structures.
      Written by:
      Date:
*/
#include <stdio.h>
#include <string.h>

// Global Structures
```

continued

PROGRAM 12-8 Demonstrate Unions in Structures *(continued)*

```
typedef struct
   {
    char  first[20];
    char  init;
    char  last[30];
   } PERSON;

typedef struct
   {
    char  type;                          // C--company: P--person
    union
       {
        char    company[40];
        PERSON  person;
       } un;
   } NAME;

int main (void)
{
// Local Declarations
   NAME business = {'C', "ABC Company"};
   NAME friend;
   NAME names[2];

// Statements
   friend.type = 'P';
   strcpy (friend.un.person.first, "Martha");
   strcpy (friend.un.person.last,"Washington");
   friend.un.person.init  = 'C';

   names[0] = business;
   names[1] = friend;

   for (int i = 0; i < 2; i++)
        switch (names[i].type)
            {
             case 'C': printf("Company: %s\n",
                             names[i].un.company);
                       break;
             case 'P': printf("Friend:  %s %c %s\n",
                             names[i].un.person.first,
                             names[i].un.person.init,
                             names[i].un.person.last);
```

continued

PROGRAM 12-8 Demonstrate Unions in Structures *(continued)*

```
                      break;
            default:  printf("Error in type\a\n");
                      break;
           } // switch
   return 0;
}  // main
```

```
Results:
Company: ABC Company
Friend:  Martha C Washington
```

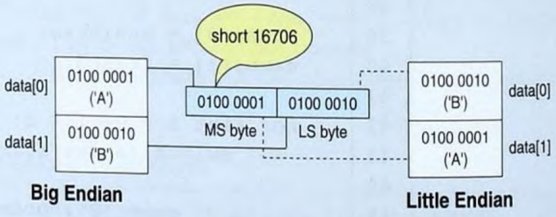
Program 12-8 Analysis We begin by creating two union structures, one for a business and one for a friend. To initialize the business *union*, we used initializers (see below). To initialize the friend *union*, we used string copy and assignment. Once the unions were created, we copied them to an array to demonstrate one common processing concept.

Unions are often processed with a **switch** statement that uses the type to determine what data format is currently being processed. Note that it is the programmer's responsibility to process the data correctly. If a business identifier is used while person data are present, the program will run but garbage will be produced.

Once the correct union type has been determined, the rest of the processing is routine. We simply print the data in the union using the appropriate identifiers for its content. Note that we provide a **default** statement even though there is no possibility of an error in this simple program.

Big and Little Endian

Computer hardware can be physically designed to store the most significant byte of a number at either the left end of the number (big endian) or at the right end of the number (little endian), as shown below.



If we consider memory as an array of bytes, then a computer that uses big endian stores the most significant byte—that is, the larger portion of the number (16,000 in the above example—at the beginning of the array. This is shown in the example on the left. Note that `data[0]` is 'A' and `data[1]` is 'B'.

In a little-endian computer, the most significant byte is stored on the right. This example is seen on the right in the above example. This time, `data[0]` contains 'B' and `data[1]` contains 'A'. If you want to know if your computer is big or little endian, then run Program 12-7 and check out the results.

Internet Addresses

As a final example of *union* processing, let's consider how Internet addresses are formatted and processed. We are most familiar with the Internet domain names such as `www.deanza.edu`. These names are easy to remember and use. The real Internet Protocol (IP) address, however, is defined as dotted-decimal notation, as shown below.

```
153.18.8.105
```

The decimal–dot notation is used primarily by system administrators. To conserve transmission time and facilitate processing, however, the IP address must be converted to a 4-byte integer for transmission down the internet.

Each portion of the IP address has a maximum value of 255. This means that it can be stored in 1 byte. To store the address in an integer, we simply convert each portion of the address to 1 byte and store it in the appropriate portion of the integer. Thus, in our example, we convert 153 to its corresponding (extended) ASCII character and store it in the first byte of the integer. Then we convert the 18 and store it in the second byte, 8 into the third byte, and finally 105 into the fourth byte. To process the integer address as both an integer and an array of four characters requires a union.

Given that the IP address is a string, we use the string token operator and a loop for the conversion. The code is shown in Program 12-9.

PROGRAM 12-9 Convert IP Address to *long*

```
/* Reformat IP string address to long integer.
      Written by:
      Date:
*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Global Structures
   typedef union
      {
       unsigned char chAddr[4];
       unsigned long numAddr;
      } IP_ADDR;

int main (void)
{
// Local Declarations
   IP_ADDR addr;
   char*   parser;
```

continued

PROGRAM 12-9 Convert IP Address to *long (continued)*

```
21       char      strAddr[16] = "153.18.8.105";
22
23    // Statements
24       // Convert first address segment
25       parser = strtok (strAddr, ".");
26       addr.chAddr[3] = strtol (parser, (char**)NULL, 10);
27
28       for (int i = 2; i>=0; i--)
29           {
30            // Convert decimal dot positions 2, 3, 4
31            parser = strtok (NULL, ".");
32            addr.chAddr[i] =
33                strtol (parser, (char**)NULL, 10);
34           } // for
35
36       // Now print results
37       printf ("IP decimal dot    : %d.%d.%d.%d\n",
38               addr.chAddr[3], addr.chAddr[2],
39               addr.chAddr[1], addr.chAddr[0]);
40       printf ("IP binary         : %lu\n", addr.numAddr);
41       return 0;
42    }  // main
```

```
Results:
IP decimal dot    : 153.18.8.105
IP binary address: 2568095849
```

ogram 12-9 Analysis

Before you study this program, you may want to review the parsing concepts discussed in Chapter 11 (see *strtok* in Chapter 11). They are used extensively in this program.

In statement 21, we set up an IP address using the dotted-decimal notation. It is constructed so that each portion is a decimal number (`A.B.C.D`). We then parse the first number and place it in the first byte of the binary address. This is necessary because the first string token format differs from the one that parses the rest of the string (see statements 25 and 31).

Within the loop, we parse the rest of the string, convert it to a decimal value, and store the address elements in the second, third, and fourth positions of the binary address. After the loop, we print the converted address, first in its character format and then as a long integer.

This program is designed to work with little-endian computers. For big endian computers statement 26 would use index 0 and the *for* in statement 28 would start at 1 and move up to 3. The indexes in statements 38 and 39 would also need to be reversed.

12.5 Programming Application

In this section, we develop a program that simulates the operation of an elevator. The elevator serves floors from zero (the basement) to the top floor. The elevator is very old and is not fully automatic. When people enter the elevator, they enter their desired floor number. Several numbers can be requested at a time. After all numbers have been entered, the passengers close the door by pressing the close door button (the return key).

Each time the door closes, the elevator checks to see if any floors in the current direction (up or down) need to be serviced. If they do, then it services these floors first, starting with the closest one to the current floor. If no floors in the current direction need service, it checks the opposite direction, again servicing the one closest to the current floor.

Each time the elevator arrives at a floor, new passengers can get on and request their floor. The new requests are added to the ones still pending, and the elevator again evaluates which floor will be processed next.

The structure for this program is portrayed in Figure 12-22. The elevator is represented as a structure with two fields: the current floor and a pointer to an array of buttons. The button values are IN, meaning the floor has been requested, and OUT, meaning the floor has not been requested. After a floor has been serviced, the button is reset.

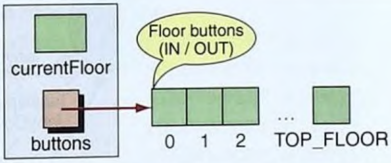


FIGURE 12-22 Elevator Structure

The elevator design is shown two different ways. Figure 12-23 is a structure chart for the program.

Figure 12-24 is a state diagram for the elevator. A state diagram is a design technique that is often used with realtime systems to show how a system moves from one state to another. For an elevator, it can be in one of three states: moving up, moving down, or stopped. Each of these states is represented by a circle in the diagram. To move from one state to another, a change must occur in the elevator environment. For example, to change from the stop state to the up state, a button must be pressed. This is reflected on the line between stop and up as anyUp.

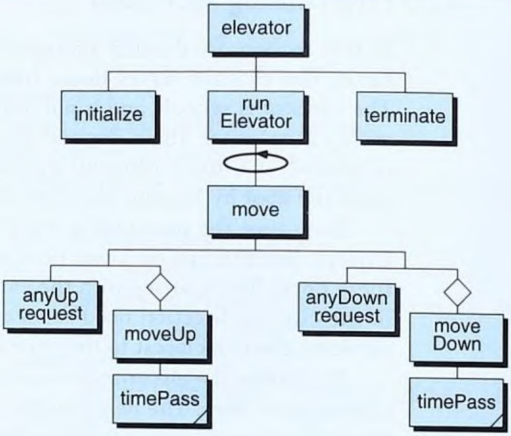


FIGURE 12-23 Elevator Structure Chart

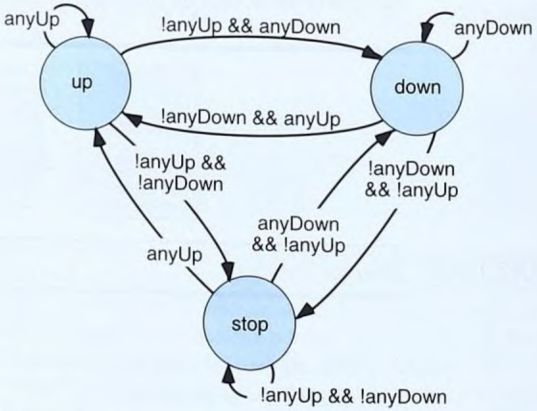


FIGURE 12-24 Elevator State Diagram

The *main* function for the elevator program is seen in Program 12-10. It calls three other functions: initialize (Program 12-11), run elevator (Program 12-12), and terminate (Program 12-17). Of the three, run elevator is the function of primary interest; it simulates the actual running of the elevator.

PROGRAM 12-10 Elevator: *main*

```
/* This program simulates the operation of an elevator.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdbool.h>

#define TOP_FLOOR 10
#define DELAY_FACTOR 10000

// Global Type Declarations
typedef enum {OUT, IN} BUTTON_STATUS;
typedef enum {DOWN, STOP, UP} DIRECTION_STATUS;
typedef struct
   {
    int            currentFloor;
    BUTTON_STATUS* buttons;
   } ELEVATOR;

// Function Declarations
void initialize      (ELEVATOR* elev);
void runElevator     (ELEVATOR* elev);
void terminate       (ELEVATOR* elev);
void move            (ELEVATOR* elev);
bool anyUpRequest    (ELEVATOR* elev);
bool anyDownRequest  (ELEVATOR* elev);
void moveUp          (ELEVATOR* elev);
void moveDown        (ELEVATOR* elev);
void timePass        (int m);

int main (void)
{
// Local Declarations
ELEVATOR elevator;

// Statements
   initialize  (&elevator);
   runElevator (&elevator);
   terminate   (&elevator);
```

continued

PROGRAM 12-10 Elevator: *main (continued)*

```
   return 0;
}  // main
```

Program 12-11 initializes the state of the elevator.

PROGRAM 12-11 Elevator: Initialize

```
/* ================== initialize ==================
   This function dynamically allocates memory locations
   for buttons & initializes current floor to 1 to show
   that the elevator is parked in the first floor.
      Pre   nothing
      Post  elevator created, all buttons reset, and
            parked at first floor (not basement)
*/
void initialize (ELEVATOR* elev)
{
// Statements
   elev->buttons = calloc(TOP_FLOOR + 1,
                          sizeof (BUTTON_STATUS));

   for (int i = 0; i <= TOP_FLOOR; i++)
      elev->buttons [i] = OUT;
   elev->currentFloor = 1;

   return;
}  // initialize
```

Program 12-12 simulates the operation of the elevator.

PROGRAM 12-12 Elevator: Run Elevator

```
/* ================= runElevator ===================
   Simulate the operation of the elevator.
      Pre   elevator structure has been initialized
      Post  simulation is complete
*/
void runElevator (ELEVATOR* elev)
{
// Local Declarations
   char  buffer [81];
   int   floor;
   char* pStrIn;

```

continued

PROGRAM 12-12 Elevator: Run Elevator *(continued)*

```
// Statements
  printf("\n\nThis elevator goes from basement (0) ");
  printf("to floor %d", TOP_FLOOR);
  printf( "\n\nType floors & press return to start");
  printf( "\nIf no new floors, press return key.");
  printf("\nTo quit, key EOF");
  printf("\n\nPlease enter floors: ");

  while (fgets(buffer, 81, stdin))
    {
     pStrIn = buffer;
     while (*pStrIn != '\n')
       {
        // Locate next floor digit & convert to int
        while (*pStrIn == ' ')
          pStrIn++;
        if (!isdigit (*pStrIn))
          {
           printf("\aInvalid floor %c\n", *pStrIn);
           pStrIn++;
          } // if
        else
          {
           sscanf (pStrIn, "%d", &floor);
           if (floor  == elev->currentFloor)
              printf("\n\aAlready on floor %d.",
                   elev->currentFloor);
           else
             if (floor < 0 || floor > TOP_FLOOR)
                printf("\n\a%d invalid floor",
                     floor);
             else
               elev->buttons [floor] = IN;

           // Synchronize sscanf & *pStrIn
           while (isdigit (*pStrIn))
             pStrIn++;
          } // else
       } // while

     move (elev);
     printf("\n\nPlease enter floors: ");
   } // while
   return;
}  // runElevator
```

Program 12-12 Analysis The simulation is controlled by `runElevator`. The technique we use to prevent invalid user input is interesting. After reading the input stream, we use a pointer to parse the string after skipping whitespace. If the current character is a digit, we use **sscanf** to format the digit(s) as an integer. If it is not a digit, we print an error message and skip it. (Of course, in a real elevator, it is impossible to get an invalid floor.) Then, after converting the digit string to an integer, we synchronize the buffer and our pointer with a small **while** loop.

As we parse the integers in the input stream, we "push" their corresponding buttons in our data structure. This code is shown at statement 45 in Program 12-12.

Program 12-13 moves the elevator to a requested floor.

PROGRAM 12-13 Elevator: Move

```
/* =================== move ===================
   Moves the elevator to a requested floor. It stops
   the elevator after responding to one request.
      Pre   given elevator
      Post  elevator has been moved--while it is
            moving, the floors are called out
*/
void move (ELEVATOR* elev)
{
// Local Declarations
   static DIRECTION_STATUS direction = STOP;

   bool anyUp;
   bool anyDown;

// Statements
   anyUp    = anyUpRequest   (elev);
   anyDown  = anyDownRequest (elev);

   if (direction == UP)
      {
       if (!anyUp && anyDown)
           direction = DOWN;
       else
           if (!anyUp && !anyDown)
               direction = STOP;
      } // UP

   else if (direction == DOWN)
      {
       if (!anyDown && anyUp)
           direction = UP;
       else
```

continued

PROGRAM 12-13 Elevator: Move *(continued)*

```
34            if (!anyDown && !anyUp)
35                direction = STOP;
36         } // DOWN
37
38      else if (direction == STOP)
39         {
40          if (anyUp)
41              direction = UP;
42          else
43              if (anyDown)
44                  direction = DOWN;
45         } // else if stop
46
47      if (direction == UP)
48          moveUp (elev);
49      else
50          if (direction == DOWN)
51              moveDown (elev);
52          else
53              printf("\n***** NO BUTTON PRESSED ***** ");
54      return;
55   }  // move
```

Program 12-13 Analysis The move function is one of the more interesting functions in the simulation. At any given point in the operation, the elevator can be in one of three states: moving up, moving down, or stopped. We use an enumerated type to track the state.

If the elevator is moving up, we continue up as long as there are requests for higher floors. If there are none, we test for down requests. If there are any down requests, we change the state to down and proceed. If there are no up requests and no down requests, we change the state to stopped. This logic is seen in statements 21 through 47 of Program 12-13. Note the use of the blocks to ensure that the **if** and **else** statements are properly paired.

Program 12-14 simulates the elevator movement as it moves up and down.

PROGRAM 12-14 Elevator: Move Up and Move Down

```
/* ================== moveUp ==================
   This function simulates the movement of the elevator
   when it is going up.
      Pre   given moving up elevator
      Post  up simulation is displayed on the screen
*/
void  moveUp (ELEVATOR* elev)
{
```

continued

PROGRAM 12-14 Elevator: Move Up and Move Down *(continued)*

```
// Statements
   printf ("\nThe door is being closed …");
   printf ("\nWe are going up.");
   (elev->currentFloor)++;
   while (elev->buttons[elev->currentFloor] != IN)
      {
       printf("\n");
       timePass (2);
      printf("\nPassing floor %d", elev->currentFloor);
       printf("\n ");
       timePass (2);
       (elev->currentFloor)++;
      } // while

   elev->buttons [elev->currentFloor] = OUT;
   printf("\nThe door is  being opened …");
   printf("\n");
   printf("\n ***** FLOOR %d ***** ",
          elev->currentFloor);
   printf("\n");
   timePass(4);
   return;
}  // moveUp

/* ==================== moveDown ====================
   This function simulates the movement of the elevator
   when it is going down.
      Pre   given moving down elevator
      Post  down simulation is displayed on the screen
*/
void moveDown (ELEVATOR* elev)
{
// Statements
   printf("\nThe door is being closed …");
   printf("\nWe are going down");
   (elev->currentFloor)--;
   while (elev->buttons [elev ->currentFloor] != IN)
      {
       printf("\n");
       timePass (2);
      printf("\nPassing floor %d", elev->currentFloor);
       printf("\n");
       timePass (2);
       (elev->currentFloor)--;
```

continued

PROGRAM 12-14 Elevator: Move Up and Move Down *(continued)*

```
      } // while
   elev->buttons [elev->currentFloor] = OUT;
   printf("\nThe door is being opened …");
   printf("\n ");
   printf("\n ***** FLOOR %d ***** ",
          elev->currentFloor);
   printf("\n");
   timePass (4);

   return;
}  // moveDown
```

Program 12-14 Analysis Both `moveUp` and `moveDown` operate similarly. They move past the current floor in the correct direction and "call out" the floors as we pass them. A timing loop is included to simulate the time it takes the elevator to reach the next floor.

Program 12-15 contains two functions: `anyUpRequest` checks for floors above the current position; `anyDownRequest` checks for floors below the current position.

PROGRAM 12-15 Elevator: Any Up and Any Down Request

```
/* ================ anyUpRequest ==================
   This function checks to see if any request is for a
   floor above the current floor.
      Pre   given elevator
      Post  return true if button above current floor
            pushed; return false otherwise
*/
bool anyUpRequest (ELEVATOR* elev)
{
// Local Declarations
   bool  isAny = false;

// Statements
   for (int check = elev->currentFloor;
        check <= TOP_FLOOR && !isAny;
        check++)
      isAny = (elev->buttons[check] == IN);
   return isAny;
}  // anyUpRequest

/* ================= anyDownRequest =================
   This function checks to see if any request is for a
```

continued

PROGRAM 12-15 Elevator: Any Up and Any Down Request *(continued)*

```
23        floor below the current floor.
24           Pre   given elevator
25           Post  return true if button below current floor
26                 pushed; return false otherwise
27     */
28     bool anyDownRequest (ELEVATOR* elev)
29     {
30     // Local Declarations
31        bool isAny = false;
32
33     // Statements
34        for (int check = elev->currentFloor;
35                 check >=  0;
36                 check--)
37           isAny = isAny || (elev->buttons[check] == IN);
38        return isAny;
39     }  // anyDownRequest
```

Program 12-15 Analysis Compare the **for** loops in `anyDownRequest` and `anyUpRequest`. One is much more efficient than the other. Do you see why? In `anyUpRequest`, we stop when we find the first request for a floor above the current one. In `anyDownRequest`, we check all floors below the current one, even if the first one has been requested. This means we always examine all lower floors.

Note the logic to set `isAny` at statement 37 in Program 12-15. The current setting is or'ed with a button. If either are true, the result is true. Thus, once we find a floor has been requested, `isAny` will remain set regardless of the settings of the other buttons. This is a common technique to analyze a series of logical values.

Program 12-16 contains a loop to simulate time passing. We use a simple *for* loop.

PROGRAM 12-16 Elevator: Time Pass

```
/* ================== timePass ==================
   This function simulates the concept of passing time by
   executing an empty for-loop.
      Pre   time to be passed (number of moments)
      Post  time has passed
*/
void timePass  (int time)
 {
// Statements
   for (long i = 0; i < (time*  DELAY_FACTOR); i++)
      ;
```

continued

PROGRAM 12-16 Elevator: Time Pass *(continued)*

```
    return;
}  // timePass
```

Program 12-16 Analysis The factor in the timing loop depends on the speed of your computer. In this case, we use 10,000. On a faster computer, you need to make the factor larger. On a slower computer, you need to make it smaller.

Note how we code the null statement in the **for** loop. Putting the null statement on a line by itself makes the function more readable. If the null statement were put at the end of the *for* statement, it would be too easy to think that the next statement, the **return** in this function, were part of the **for** loop.

The termination code is shown in Program 12-17.

PROGRAM 12-17 Elevator: Terminate

```
/* ================= terminate =================
   Release the memory occupied by buttons.
      Pre   given elevator
      Post  elevator memory is released
*/
void terminate (ELEVATOR* elev)
{
// Statements
   free (elev->buttons);
   return;
}  // terminate
// ================= End of Program =================
```

Program 12-17 Analysis Although we do not need to release memory when terminating a program, we include the logic to do so for completeness.

12.6 Software Engineering

In this section we discuss two important aspects of program design: function coupling and data hiding.

Coupling

In Chapter 11, we discussed a concept known as functional cohesion, a measure of how closely related the processes are within a function. A related topic, coupling, is a measure of how tightly two functions are bound to each other. The more tightly coupled they are, the less independent they are. Since our objective is to make the modules as independent as possible, we want them to be loosely coupled.

Loose coupling is desirable for several reasons:

1. Independent—that is, loosely coupled—functions are more likely to be reusable.
2. Loosely coupled functions are less likely to create errors in related functions; conversely, the tighter the coupling, the higher the probability that an error in one function will generate an error in a related function.
3. Maintenance modifications—that is, modifications required to implement new user requirements—are easier and less apt to create errors with loosely coupled functions.

In his book on designing structured systems, Page-Jones describes five types of coupling. We review them here. For an in-depth discussion of the concept, refer to Chapter 5 in Page-Jones's book, *The Practical Guide to Structured Systems Design*.[2]

Data Coupling

Data coupling passes only the minimum required data from the calling function to the called function. All required data are passed as parameters, and no extra data are passed. This is the best form of coupling and should be used whenever possible.

When you write simple functions that work on only one task, the coupling naturally tends to be data coupling. Consider, for example, the function `exchange` in the selection sort in Program 10-4. This function exchanges two integers. It receives pointers to the two integers that it will exchange and nothing else. It makes no references to any data outside the function, except through the parameter pointers. This function uses data coupling and is highly reusable.

Well-structured functions are highly cohesive and loosely coupled.

[2] Meilir Page-Jones, *The Practical Guide to Structured Systems Design*, Yourdon Press Computing Series (Upper Saddle River, N.J.: Prentice Hall, 1988).

We could have fallen into the trap of passing extra parameters by passing the function, the array, and the index locations of the two integers to be exchanged. The function would have worked just as well, but the coupling would not have been as loose. Now it requires an array of integers instead of just integers. Furthermore, we could have made the coupling even tighter had we referred to the maximum size of the array using the precompiler declaration `SIZE`. At this point, it is highly questionable whether the function could be used in another program.

Stamp Coupling

Functions are **stamp coupled** if the parameters are composite objects such as arrays or structures. Most of the functions in the selection sort in Chapter 10 use stamp coupling because they pass the array. (Although it could be argued that we are passing only a pointer to the array, the intent is to modify the array. We are, therefore, passing the array for the purposes of this discussion.)

You should now be arguing, "But we have to pass the array!" Yes, that is true. Stamp coupling is not bad and is often necessary. The danger with stamp coupling is that often it is just too easy to send a structure when all the data in the structure are not required. When extra data are sent, we begin to open the door for errors and undesired side effects.

Consider the time stamp described in Figure 12-15. This structure contains two nested structures, `date` and `time`. If we were to use these data, for example, to print the date in a report heading, and passed the whole structure, we would send too much data! In addition, if we were to pass the structure by address rather than by value, we risk the possibility of an error in one function accidentally changing the data in the structure and causing a second error. The correct solution is to pass only the data that are needed and then only by value when possible.

Stamp coupling should pass only the data needed.

A common practice to reduce the number of parameters required for a function is to create a structure that contains all the data the function needs and pass it. Page-Jones refers to this as *bundling.*[3] It is a common practice, but it is not a good practice for three reasons.

1. Maintenance is made more difficult because it is more difficult to trace data through a program.
2. Extra data can be passed. For example, a bundled structure is created for a series of related functions, but not all of them use all the data. The temptation is just too great to pass the structure even though only one or two of the members are needed.
3. The semantics of the structure are often artificial, making the program more difficult to read and understand.

[3] Meilir Page-Jones, *The Practical Guide to Structured Systems Design.*

Don't bundle unrelated data to reduce the number of parameters.

Control Coupling

Control coupling is the passing of flags that may be used to direct the logic flow of a function. It closely resembles data coupling except that a flag is being passed rather than data.

In C, flags are often returned from a function rather than being passed as parameters, but the intent and usage are the same. For example, consider the return values from **scanf.** It returns either `EOF`, a definite flag, or the number of values successfully read, which can also be used as a flag for success. An example of a flag being passed in a function you might write is the user-selected option in the menu function of an interactive program. This flag directs the entire flow of the program. The option is a special type of flag known as a data flag. It is data entered by the user and at the same time it is a flag intended to direct the flow of the program.

Properly used, control coupling is a necessary and valid method of communicating between two functions. Like stamp coupling, however, it can be misused. Properly used, it communicates status: The end of the file has been reached. The search value was found.

Poor flag usage is usually an indication of poor program design, for example, when a process is divided between two or more independent functions. Flags used to communicate horizontally across several functions in the structure chart are often an indication of poor design. Action flags, as opposed to status flags, that require the receiving function to perform some special processing are also highly suspect. An example of an action flag is a flag that directs a customer's purchase not be approved rather than simply reporting that the credit limit has been exceeded or that no payment was received last month.

Control coupling should be used only to pass status.

Global Coupling

Global coupling uses global variables to communicate between two, or usually more, functions. Page-Jones calls it *common coupling*. With all that we have said about not using global variables, it should not come as a surprise that this is not a good coupling technique. In fact, it should *never* be used.

You should never use global coupling for several reasons. We cite only the big three.

1. Global coupling makes it virtually impossible to determine which modules are communicating with each other. When a change needs to be made to a program, therefore, it is not possible to evaluate and isolate the impact of the change. This often causes functions that were not changed to suddenly fail.
2. Global coupling tightly binds a function to the program. This means that it cannot be easily transported to another program.

3. Global coupling leads to multiple flag meanings. This problem is often made worse by using generic flag names, such as `f1`, `f2`, ..., `f21`. (Finding 21 flags in a single program is not an exaggeration. We know of one assembly program that had more flags. In fact, it had one flag that was used solely to indicate that another flag had been set but was now turned off; in other words, a flag that returned the status of a flag!)

   The danger here should be obvious. If a flag can be used globally to communicate between two functions, it is highly probable that at some point this flag could be erroneously changed by a third function that used it for another purpose.

Avoid global coupling within a program.

Content Coupling

The last type of coupling is very difficult, but not impossible, to use in C. **Content coupling** occurs when one function refers directly to the data or statements in another function. Obviously, this concept breaks all the tenets of structured programming.

Never use content coupling.

Referring to the data in another function requires that the data be made externally visible outside the function. This is impossible in C. The only thing that comes even remotely close is global variables. Since we have stressed the dangers of global variables before, we will simply state here that they should not be used for communication within one compile unit.

Data Hiding

We have previously discussed the concept of global and local variables. In the discussion, we pointed out that anything placed before *main* was said to be in the global part of the program. With the exception of data that need to be visible to functions in other compile units, no data need to be placed in this section.

One of the principles of structured programming states that the internal data structure should be hidden from the user's view. The two terms you usually hear are *data hiding* and *data encapsulation*. Both of these principles have as their objective protecting data from accidental destruction by parts of your program that don't need access to the data. In other words, if a part of your program doesn't need data to do its job, it shouldn't be able to *see* the data.

Programming Standard:
Do not place any variables in the global area of a program.

Any variables placed in the global area of your program—that is, before *main*—can be used and changed by every part of your program. This is in direct conflict with the structured programming principles of data hiding and data encapsulation.

Summary

We have described five different ways that two functions can communicate. The first three are all valid and useful, although not without some dangers. These communication techniques also provide data hiding. Data coupling is universally accepted and provides the loosest communication between two functions. Stamp and control coupling present some dangers that must be recognized. When using stamp coupling, do not pass more data than are required. Keep control coupling narrow—that is, between only two functions. You should always avoid the last two, global and content coupling. They do not protect the data.

12.7 Tips and Common Programming Errors

1. Don't forget the semicolon at the end of the declaration of structures and unions. This is one of the locations where you see a semicolon after a closing brace (}) in C.

2. Because the direct selection operator has a higher precedence than the indirection operator, parentheses are required to reference a member with a pointer.

```
(*ptr).mem
```

3. The indirect selection operator (->) is one token. Do not put a space between its symbols (between - and >).

4. To access a member in an array structure, you need the index operator. For example, the correct expression to access a member named mem in an array of structure named ary is

```
ary[i].mem
```

5. The type name in a **typedef** comes after the closing brackets and before the semicolon:

```
typedef  struct
    {
    …;
    …;
    } TYPE_NAME;
```

6. You cannot define a variable at the same time that you declare a type definition. In other words, you are *not* allowed to do the following:

```
typedef struct
    {
    …;
    …;
    } TYPE_NAME variable_name;                    // ERROR
```

7. You may not use the same structure name with a tag inside a structure. The following declaration is not valid:

```
typedef struct TAG_NAME
    {
    …;
    …;
    struct TAG_NAME field_name;                   // ERROR
    } ID;
```

8. A union can store only one of its members at a time. You must always keep track of the available member. In other words, storing one data type in a union and accessing another data type is a logic error and may be a serious run-time error.

9. Although structures can be assigned to each other, it is a compile error to assign a structure of one type to a structure of a different type.

10. It is a compile error to use a structure type with only its tag; that is, without the keyword **struct,** as shown below.

```
struct stu
     {
      ...
     };
stu aStudent;                             // Compile Error
struct stu aStudent;                      // Correct Code
```

11. It is a compile error to compare two structures or unions, even if they are of the same type.

12. It is a compile error to refer to a structure member without qualification, such as `id` rather than `student.id`.

13. It is a compile error to omit a level of qualification when structures are nested. For example, in Figure 12-13, it is an error to omit the time qualifier in the following reference.

```
stamp.time.min
```

14. It is a compile error to forget the address operator with **scanf** when referring to a nonstring member. The pointer is the address of the structure, not the address of the member.

15. Referencing an identifier in a union when a different type is active is a logic error and may cause the program to fail.

16. It is a compile error to initialize a union with data that do not match the type of the first member.

12.8 Key Terms

content coupling
control coupling
data coupling
enumeration constant
global coupling
nested structures

slack bytes
stamp coupling
structure
structure variable
tagged structure
union

12.9 Summary

❑ An enumerated type is built on the standard type, integer.

❑ In an enumerated type, each identifier is given an integer value.

❑ A structure is a collection of related elements, possibly of different types, having a single name.

❑ Each element in a structure is called a field.

❑ One difference between an array and a structure is that all elements in an array must be of the same type while the elements in a structure can be of the same or different types.

❑ We have discussed two different ways to declare and/or define a structure: tagged structure and type-defined structure.

❑ A structure can be initialized when it is defined. The rule for structure initialization is the same as for array initialization.

❑ We can access the members of a structure using the direct selection operator (`.`).

❑ Structures can be accessed through a pointer. The best way to do this is to use the indirect selection operator (->).

❑ The following two expressions are the same if `ptr` is a pointer to a structure:

```
(*p).x          p->x
```

❑ The information in a structure can be sent to a function using one of the following methods:

1. Sending individual members
2. Sending the whole structure
3. Sending a pointer to the structure

❑ A union is a construct that allows a portion of memory to be used by different types of data.

❑ In software engineering, coupling is the measure of how tightly two functions are bound to each other.

❑ Computer science has identified five types of coupling: data, stamp, control, global, and content.

❑ Functions in a well-structured program are loosely coupled.

❑ Avoid global coupling and never use content coupling.

❑ Good program design can be measured by three principles: Modules must be independent, modules must be loosely coupled, and each module must do a single job.

12.10 Practice Sets

Review Questions

1. Variables are created with a type definition (*typedef*).
   a. True
   b. False
2. An integer value can be assigned to only one enumeration constant in an enumerated type.
   a. True
   b. False
3. A structure variable is used to declare a type containing multiple fields.
   a. True
   b. False
4. The indirect selection operator is used with a pointer to access individual fields in a structure.
   a. True
   b. False
5. The structured programming concept known as coupling describes how data are passed to functions.
   a. True
   b. False
6. Which of the following is not a derived type?
   a. Arrays
   b. Enumerated
   c. Float
   d. Pointers
   e. Union
7. The ______________ can be used to create a new type that can be used anywhere a type is permitted.
   a. array
   b. record type
   c. structure (*struct*)
   d. type definition
   e. both a structure and a type definition
8. The enumerated type (*enum*) is derived from the ______________ type.
   a. character
   b. boolean
   c. floating-point
   d. integer
   e. structured

9. Which of the following statements about enumerated types is true?
   a. Declaring an enumerated type automatically creates a variable.
   b. Declaring an enumerated variable without a tag creates an enumerated type.
   c. Enumerated types cannot be used in a type definition.
   d. The enumerated values are automatically assigned constant values unless otherwise directed.
   e. The identifiers in an enumerated type are enumeration variables.

10. A(n) ______________ is the smallest element of named data that has meaning.
   a. array
   b. field
   c. record type
   d. structure (*struct*)
   e. type

11. Which of the following statements about structures (*struct*) is true?
   a. A structure without a tag can be used as a derived type.
   b. Structures are derived types based on the integer standard type.
   c. The fields in a structure must be of the same type.
   d. C normally uses two different structure declarations: tagged and type-defined.

12. The ______________ structure creates a type that can be used later to define variables.
   a. array
   b. record-typed
   c. tagged
   d. type-defined
   e. variable

13. Given a structure variable named `stu`, whose type is `STU`, which contains a field named `major`, which of the following statements correctly refers to `major`?
   a. `major`
   b. `stu-major`
   c. `stu.major`
   d. `STU-major`
   e. `STU.major`

14. Given a pointer, `ptr`, to a structure `stu` containing a field `name`, which of the following statements correctly references `name`?
   a. `ptr.name`
   b. `ptr->name`
   c. `ptr.stu.name`
   d. `ptr->stu->name`
   e. `ptr->stu.name`

15. A(n) ________________ is a construct that allows a portion of memory to be shared by different types of data.

    a. *array*
    b. *field*
    c. *struct*
    d. *union*
    e. *variable*

16. Determine which of the following statements are true and which are false:

    a. A structure cannot have two fields with the same name.
    b. A structure cannot have two fields of the same type.
    c. A structure must have at least one field.
    d. A field in a structure can itself be a structure.

17. Determine which of the following statements are true and which are false:

    a. A *union* can have another *union* as one of the fields.
    b. A structure cannot have a *union* as one of its fields.
    c. A structure cannot have an array as one of its elements.
    d. When accessing the fields of a structure through a pointer p, the following two expressions are the same:

```
(*p).field_name    p->field_name
```

Exercises

18. Declare a tagged structure for a student record consisting of five fields: student ID (integer), first name (a dynamically allocated string), last name (a dynamically allocated string), total credits completed (integer), and accumulated grade point average (floating-point). A graphic representation of the structure is shown in Figure 12-25.

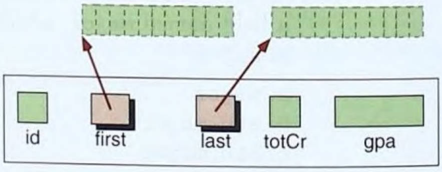


FIGURE 12-25 Data for Exercise 18

19. Declare a type-defined structure for an inventory item consisting of six fields: part number (integer), part description (a dynamically allocated string), reorder point (integer), number of items currently on hand (integer), unit measure (a string, maximum size 8), and unit price

(floating-point). A graphic representation of the structure is shown in Figure 12-26.

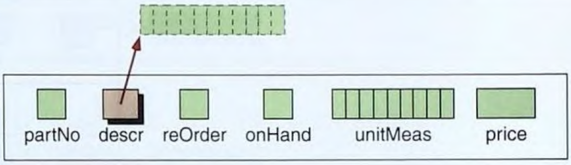


FIGURE 12-26 Data for Exercise 19

20. Declare an array of 12 elements. Each element is a structure with three fields. The first field shows the month in numeric form (1 to 12). The second field shows the name of the month (a dynamically allocated string). The third field shows the number of days in the month. Initialize the array. A graphic representation of the structure is shown in Figure 12-27.

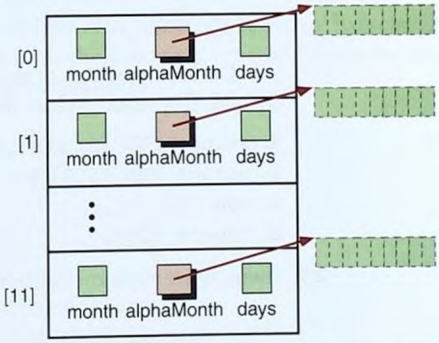


FIGURE 12-27 Data for Exercise 20

21. Declare a calendar as an array of 366 elements. Each element of the array is a structure having three fields. The first field is the name of the month (a dynamically allocated string). The second field is the day in the month (an integer). The third field is the description of activities for a particular day (a dynamically allocated string). A graphic representation of the structure is shown in Figure 12-28.

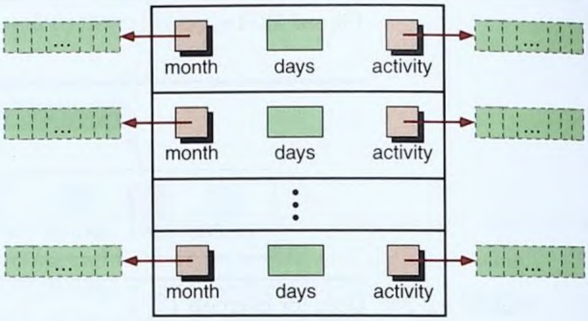


FIGURE 12-28 Data for Exercise 21

22. Imagine we have declared the following structure:

```
typedef struct
   {
    char  x;
    char* y;
    int   z[20];
   } FUN;
```

Determine which of the following definitions are valid and which are invalid. If invalid, explain the error.

a. `struct FUN f1;`
b. `struct FUN f5[23];`
c. `FUN f3;`
d. `FUN f4 [20];`

23. Imagine we have the following declaration and definitions:

```
struct FUN
   {
    char  x;
    char* y;
    int   z[20];
   } ;

struct FUN fn1;
struct FUN fn2;
struct FUN fn3 [10];
struct FUN fn4 [50];
```

Determine which of the following assignment statements are valid and which are invalid. If invalid, explain the error.

a. `fn1.x = 'b';`
b. `fn2.y = 'b';`
c. `fn3[4].z[5] = 234;`
d. `fn4[23].y = "1234";`
e. `fn4[23] = fn3[5];`

24. Imagine we have the following declaration:

```
typedef enum {ONE = 1, TWO = 2} CHOICE;
typedef union
   {
    char  choice1;
    int   choice2;
   } U_TYPE;

typedef struct
   {
    float   fixedBefore;
    CHOICE  choice;
    U_TYPE  flexible;
    float   fixedAfter;
   } S_TYPE;
```

Draw a schematic diagram for S_TYPE.

25. Using the declaration of S_TYPE (declared in Exercise 24), show what will be printed from the following program segment. (Assume that the S_TYPE declaration is global.)

```
#include <stdio.h>
int main (void)
{
 S_TYPE  s;
 S_TYPE* ps;

 s.fixedBefore = 23.34;
 s.choice = ONE;
 s.flexible.choice1 = 'B';
 s.fixedAfter = 12.45;
 ps = &s;
 printf ("\n%f", ps->fixedAfter);
 printf ("\n%d", ps->flexible.choice1);
 printf ("\n%f", s.fixedBefore);
 return 0;
} // main
```

Problems

26. Write a function called elapsedTime with two parameters, the start time and the end time. Each parameter is a structure with three fields showing

the hours, minutes, and seconds of a specific time (see Figure 12-13). The function is to return a time structure containing the time elapsed between the two parameters. You must handle the situation when the start time is in the previous day.

27. Write a function called `increment` that accepts a date structure with three fields. The first field contains the month (a pointer to a string). The second field is an integer showing the day in the month. The third field is an integer showing the year. The function increments the date by 1 day and returns the new date. If the date is the last day in the month, the month field must also be changed. If the month is December, the value of the year field must also be changed when the day is 31. A year is a leap year if

    a. It is evenly divisible by 4 but not by 100, or
    b. It is evenly divisible by 400.

28. Write a function called `futureDate`. The function is to use two parameters. The first parameter is a structure containing today's date (as defined in Problem 27). The second parameter is an integer showing the number of days after today. The function returns a structure showing the next date, which may be in a future year.

29. Write a function called `later` that receives two date parameters, compares the two dates, and returns true (1) if the first date is earlier than the second date and false (0) if the first date is later. Each parameter is a structure containing a date (as defined in Problem 27).

30. Write a function that accepts an integer representing money in dollars (no fraction) and returns a structure with 6 fields. The fields represent, respectively, the minimum number of $100, $50, $20, $10, $5, and $1 bills needed to total the money in the parameter.

31. Write a function that compares two fraction structures (see Figure 12-3). If the fractions are equal, it returns zero. If the fraction in the first parameter is less than the fraction in the second parameter, it returns a negative number. Otherwise, it returns a positive number. Hint: Convert the fraction to a floating-point number.

32. A point in a plane can be represented by its two coordinates, x and y. Therefore, we can represent a point in a plane by a structure having two fields as shown below.

```
typedef struct
{
 int  x;
 int  y;
} POINT;
```

    Write a function that accepts the structure representing a point and returns an integer (1, 2, 3, or 4) that indicates in which quadrant the point is located, as shown in Figure 12-29. Zero is positive.

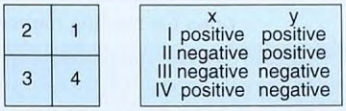


FIGURE 12-29 Quartile Coordinates for Problem 32

33. A straight line is an object connecting two points. Therefore, a line can be represented by a nested structure having two structures of the type `POINT`, as defined in Problem 32.

```
typedef struct
   {
    POINT  beg;
    POINT  end;
   } LINE;
```

Write a function that accepts two parameters of type `POINT` and returns a structure of type `LINE` representing the line connecting the two points.

34. Write a function that accepts a structure of type `LINE` (see Problem 33) and returns an integer (1, 2, 3), where 1 means vertical, 2 means horizontal, and 3 means oblique. A vertical line is a line whose `x` coordinates are the same. A horizontal line is a line whose `y` coordinates are the same. An oblique line is a line that is not vertical or horizontal.

35. Write a function that shuffles a deck of cards. The deck of cards is represented by an array of 52 elements. Each element in the array is a structure for one card, as shown below.

```
typedef struct
   {
    char* suit;     // Clubs, Diamonds, Hearts, Spades
    int   value;    // Ace, 2..9, Jack, Queen, King
   } CARD;

typedef CARD DECK [52];
```

The function must use a random number (see Chapter 4) to ensure that each shuffle results in a different card sequence. Hint: Generate a random number in the range 0…51, and then exchange the current card with the card in the random position.

36. Program 12-8 creates a union of company and person names. Write a program that reads names from the keyboard and places them into an array of names. For each entry, prompt the user to enter a code that indicates if the name is a company or person name. After all names have

been entered, print the array. Your program is to contain separate functions for reading names and printing the array.

37. Modify the program in Problem 36 to add a function to sort the names after they have been entered. Person names are to be sorted on last name, first name, and initial.

Projects

38. Write a program to keep records and perform statistical analysis for a class of students. For each student, we need a name of up to 20 characters, an ID for four digits, four quizzes, and one examination. The student data will be stored in an array of student structures. Provide for up to 50 students.

The input is read from a text file. Each line in the file contains a student's name, four quiz scores, and one examination score in order. If a quiz or examination was not taken, the score is zero. The student's name, the quiz scores, and the examination score are all separated from each other by one or more spaces. A new line ends the data for one student. The number of lines in this file is the same as the number of students.

The output consists of a listing of the students in the order they are read from the file; no sorting is required. Print each student on a separate line with an appropriate caption for each column. After the last student, print the highest, lowest, and average score for each quiz and the examination. In determining the lowest score, do not consider zero scores. A suggested report layout is shown in Table 12-1.

			DATA			
Name	**ID**	**Quiz 1**	**Quiz 2**	**Quiz 3**	**Quiz 4**	**Exam**
Student 1	1234	23	19	22	23	89
Student 2	4321	0	23	21	18	76
...						
Student *n*	1717	21	22	18	19	91
			STATISTICS			
Highest scores		23	25	23	25	96
Lowest scores		17	15	12	18	53
Average scores		21.3	20.1	19.8	21.1	81.3

TABLE 12-1 Sample output for Project 38

The data for the project are shown in Table 12-2.

Name	ID	Quiz 1	Quiz 2	Quiz 3	Quiz 4	Exam
Julie Adams	1234	52	7	100	78	34
Harry Smith	2134	90	36	90	77	30
Tuan Nguyen	3124	100	45	20	90	70
Jorge Gonzales	4532	11	17	81	32	77
Amanda Trapp	5678	20	12	45	78	34
Lou Mason	6134	34	80	55	78	45
Sarah Black	7874	60	100	56	78	78
Bryan Devaux	8026	70	10	66	78	56
Ling Wong	9893	34	09	77	78	20
Bud Johnson	1947	45	40	88	78	55
Joe Giles	2877	55	50	99	78	80
Jim Nelson	3189	82	80	100	78	77
Paula Hung	4602	89	50	91	78	60
Ted Turner	5405	11	11	0	78	10
Evelyn Gilley	6999	0	98	89	78	20

TABLE 12-2 Data for Project 38

39. Rework Project 38 to report the average quiz score, total quizzes score, and total score for each student. Then assign a grade based on an absolute scale of 90% for A, 80% for B, 70% for C, and 60% for D. Any score below 60% is an F. A total of 500 points are available. Print the student data to the right of the input data. At the end of the report, print the number of students who earned each grade, A to F.

40. Write a program that uses an array of student structures to answer inquiries. Using a menu-driven user interface, provide inquiries that report a student's scores, average, or grade based on an absolute scale (90% A, 80% B, etc.). A fourth menu option provides all data for a requested student, and a fifth prints a list of student IDs and names. To create the array, load the data from Project 38.

41. Using a sort of your choice, modify Project 40 to sort the data on student `ID`.

42. A standard deck of playing cards consists of 52 cards as shown in Table 12-3. Create an array of 52 structures to match a new deck of cards as shown in Table 12-3.

Suit	Rank
Clubs	Ace...King
Diamonds	Ace...King
Hearts	Ace...King
Spades	Ace...King

TABLE 12-3 The Order of a New Deck of Playing Cards

Then simulate shuffling the deck using Algorithm 12-1.

ALGORITHM 12-1 Shuffle Deck of Cards

```
Algorithm shuffle
 1 loop through each card in the deck
   1  get a random number in range 0…51
   2  Swap current card with card at random position
 2 end loop
end shuffle
```

After the cards have been shuffled, print them.

43. Write a function that calculates the area of one of the geometric figures shown in Table 12-4. The function receives one parameter, a structure that contains the type of figure and the size of the components needed for the calculation structured as a union. The format of the structure is shown in Table 12-4. Use an enumerated type for the figure type.

Figure Type	Components		
Rectangle	length	width	
Circle	width		
Triangle	side1	side2	side3

TABLE 12-4 Geometric Figure Components

The formulas for the figures are shown below.

$$\mathbf{RectangleArea} = \mathbf{length} \times \mathbf{width}$$

$$\mathbf{CircleArea} = \pi \mathbf{r}^2$$

$$\mathbf{TriangleArea} = \sqrt{\mathbf{t(t-side1)\times t(t-side2)\times t(t-side3)}}$$

$$\mathbf{where\ t} = \frac{1}{2}(\mathbf{side1 + side2 + side3})$$

Then write an interactive test driver that prompts the user for the type of figure to be entered and the appropriate components. It then calls the area function and prints the area before looping for the next input. Hint: Use a *switch* statement to process the different codes.

44. Modify Program 12-5 to include functions to add, subtract, divide, and compare fractions. Also change the print function to print the fraction in its simplified format—that is, to reduce the numerator and denominator by the largest common factor. Thus, the fraction

$$\frac{20}{8}$$

would be printed as

$$\frac{5}{2}$$

The compare function is to compare the functions algebraically. Thus, if the following two fractions were compared, it would return equal:

$$\frac{20}{8} \text{ and } \frac{5}{2}$$

Then modify *main* to include a loop that allows the user to enter two fractions and an option to add, subtract, multiply, divide, or compare the fractions.

Chapter 13
Binary Input/Output

In Chapter 7, we discussed text input/output. We introduced text streams and formatting text input and output. We also discussed some text input and output facilities in Chapter 11.

In this chapter, we discuss binary input/output. We first show the difference between a text stream and a binary stream. We then discuss the C input and output functions, which are mainly designed to be used with binary streams. We conclude with a discussion of converting text files to binary files and binary files to text files.

Objectives

❏ To understand the differences between text and binary files

❏ To write programs that read, write, and/or append binary files

❏ To be able to detect and take appropriate action when file errors occur

❏ To be able to process files randomly

❏ To be able to create text files from binary files and vice versa

❏ To understand and be able to implement file merges

❏ To understand and be able to implement the classic sequential file update

13.1 Text versus Binary Streams

In Chapter 7, we described inputting data from a source and outputting data to a destination: text input/output and binary input/output. For text input/output, we convert a data type to a sequence of characters. For binary input/output, we transfer data without changing their memory representations.

We also discussed that some data sources and destinations are capable of only reading and writing text. For example, a keyboard and a monitor can be a destination for only text; a keyboard is not capable of handling binary input and output. On the other hand, a file can be used to read or write any type of data, both text and binary.

Text and Binary Files

Although a file records data only in a binary format, we can distinguish between a text file and a binary file.

Text Files

A text file is a file in which data are stored using only characters; a text file is written using a text stream. Non-character data types are converted into a sequence of characters before they are stored in the file. When data are input from a text file, they are read as a sequence of characters and converted into the correct internal formats before being stored in memory. We read and write text files using the input/output functions that convert the characters to data types as shown in Figure 13-1.

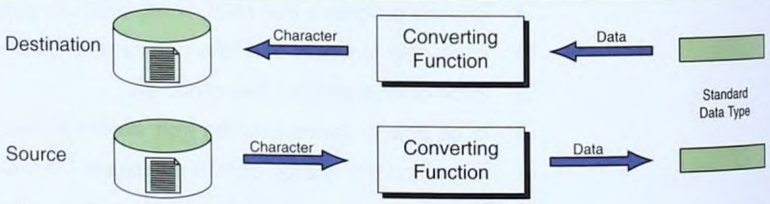


FIGURE 13-1 Reading and Writing Text Files

The converting functions consist of three categories: formatting (Chapter 7), character (Chapter 7), and string (Chapter 11). The formatting functions, such as *scanf* and *printf*, reformat a series of input characters into standard types or reformat standard type data into characters for output. The character input and output functions, such as *getchar* and *putchar*, input or output one character at a time. The string functions, such as *fgets* and *fputs*, input and output strings of characters.

Formatted input/output, character input/output, and string input/output functions can be used only with text files.

Binary Files

A binary file is a collection of data stored in the internal format of the computer. Unlike text files, the data do not need to be reformatted as they are read and written; rather, the data are stored in the file in the same format that they are stored in memory. Binary files are read and written using binary streams known as block input/output functions. Figure 13-2 depicts the operation of block read and write functions.

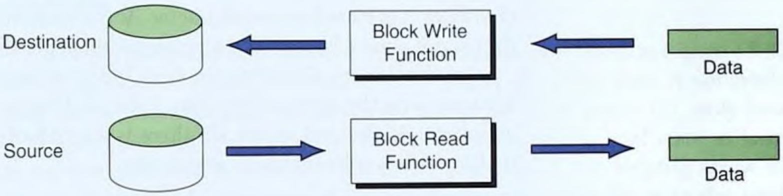


FIGURE 13-2 Block Input and Output

Differences between Text and Binary Files

Let's review the major differences between text files and binary files.

The major characteristics of text files are:

1. All data in a text file are human-readable graphic characters.
2. Each line of data ends with a newline character.
3. There may be a special character called end-of-file (EOF) marker at the end of the file.

Binary files, on the other hand, store data in their internal computer format. This means that an *int* in C is stored in its binary format, usually 4 bytes in a PC; a character is stored in its character format, usually 1 byte; and so forth. There are no lines in a binary file but there is an end-of-file marker.

The major characteristics of binary files are:

1. Data are stored in the same format as they are stored in memory.
2. There are no lines or newline characters.
3. There may be an end-of-file marker.

Figure 13-3 shows how two data items are stored in a text file (which uses 8-bit ASCII code) and in a binary file.

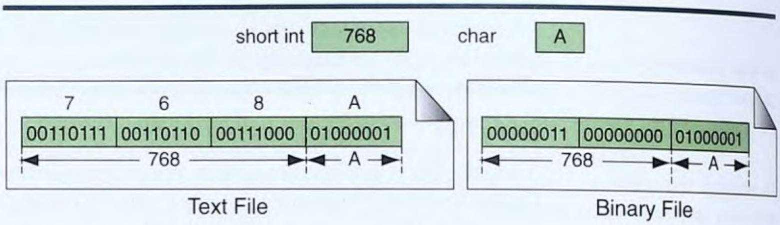


FIGURE 13-3 Binary and Text Files

In the text file, the number 768 is stored as three numeric characters. The character A is stored as one character. At the end, there is an end-of-file marker that is not shown because its representation depends on the implementation.

In the binary file, the number 768 can be stored in 2 bytes, which assumes that the *short int* type has a size of 2 bytes. The character A is stored as 1 byte. At the end of the file there is an **end-of-file** (EOF) **marker**, again its format is implementation dependent.

Text files store data as a sequence of characters; binary files store data as they are stored in primary memory.

State of a File

An opened file is either in a read state, a write state, or an error state. If we want to read from a file, it must be in the **read state**. If we want to write to a file, it must be in the **write state**. The **error state** results when an error occurs when the file is opened or when it is read or written. If we try to read from a file in the write state, an error occurs. Likewise, if we try to write to a file in the read state, an error occurs. When a file is in an error state, we cannot read from it or write to it.

To open a file in the read state, we specify the read mode in the open statement. A file opened in the read state must already exist. If it doesn't then the open fails.

A file can be opened in the write state using either the write or append mode. In write mode, the writing starts at the beginning of the file. If the file already exists, its data are lost. In append mode, the data are added at the end of the file. If the file exists, the writing begins after the existing data. If the file does not exist, a new file is created.

In addition to read, write, and append, files can be opened in the **update mode**. Updating allows a file to be both read and written. Even when the file is opened for updating, however, it can still be in only one file state at a time.

To open a file for updating, we add a plus sign (+) to the basic mode. The initial state of a file opened for updating is determined by the basic mode: `r+` opens the file in the read state, while `w+` and `a+` open the file in the write state. Table 13-1 lists the six file modes with their features.

Mode	r	w	a	r+	w+	a+
Open State	read	write	write	read	write	write
Read Allowed	yes	no	no	yes	yes	yes
Write Allowed	no	yes	yes	yes	yes	yes
Append Allowed	no	no	yes	no	no	yes
File Must Exist	yes	no	no	yes	no	no
Contents of Existing File Lost	no	yes	no	no	yes	no

TABLE 13-1 File Modes

The file states and their potential error conditions are shown in Figure 13-4. Study this figure carefully. If a state is shaded gray, then that state is not available for the indicated file mode. As shown in the read mode (r), only two states are possible, read and error. The file will stay in the read state as long as we use only read functions. That is the meaning of the looping arrow in the read-state circle. However, if we try to write when the file is in the read state, the state changes to the error state. Once the file is in an error state, any subsequent attempt to read it will result in an error.

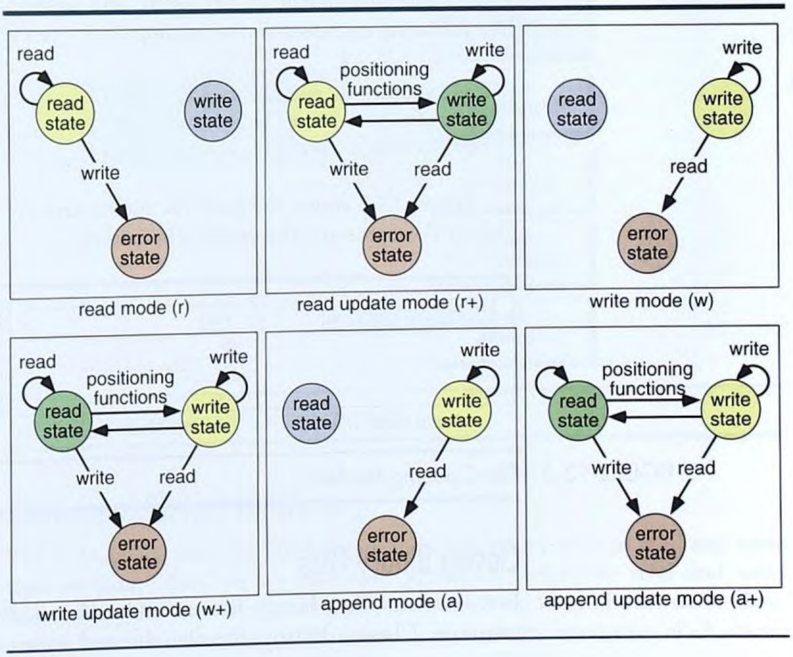


FIGURE 13-4 File States

Now look at the read update mode, r+. In this mode, the file can be in either the read or the write state. To move from one to the other, we must use one of the positioning functions, which are discussed later. That is the meaning of the arrows between the read state and write state. If we try to write after a read without repositioning, the file will be in the error state. The error can be cleared by *clearerr,* which is discussed in the section "File Status Functions" later in the chapter.

Opening Binary Files

The basic open operation is unchanged for binary files—only the mode changes. The function declaration for *fopen* is repeated here for convenience.

```
FILE* fopen (const char* filename, const char* mode);
```

Recall that the file name is the name used for the file by the operating system. The name we use internally in our program is a file pointer whose address is filled by the *fopen* return value. Here, we are most interested in the mode, which is a string that contains a code defining how the file is to be opened—for reading, writing, or both—and its format—text or binary. Text files use just the basic modes discussed above. To indicate that a file is binary, we add a binary indicator (b) to the mode. The six binary file modes are: read binary (rb), write binary (wb), append binary (ab), read and update binary (r+b), write and update binary (w+b), and append and update binary (a+b). The following examples contain examples of open statements for binary files.

```
spReadBin = "myFile.bin", "rb");      // Read Binary
spWriteUp = "myFile.bin", "w+b");     // Write with Update
spApndBin = "myFile.bin", "ab");      // Append Binary
```

Figure 13-5 shows the basic file modes and the file state they create. The marker (s) represents the end-of-file marker.

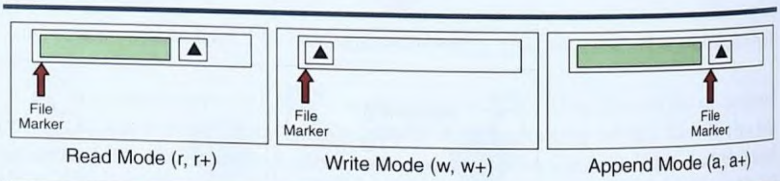


FIGURE 13-5 File-Opening Modes

Closing Binary Files

Just like text files, binary files must be closed when they are not needed anymore. Closing destroys the file table and erases the logical file name. The close function was covered in Chapter 7. Its function declaration is repeated here for your convenience.

```
int fclose (FILE* sp);
```

13.2 Standard Library Functions for Files

C has eight categories of standard file library functions (see Figure 13-6). We have already discussed the first four in Chapter 7 and Chapter 11; open and close, character input and output, formatted input and output, and line input and output. We discuss the other four categories, which are more related to binary files, in this section.

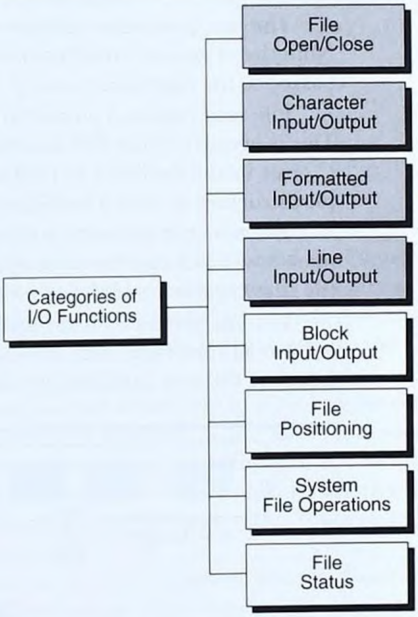


FIGURE 13-6 Types of Standard Input/Output Functions

Block Input/Output Functions

The C language uses the block input and output functions to read and write data to binary files. As we discussed previously, when we read and write binary files, the data are transferred just as they are found in memory. There are no format conversions. This means that, with the exception of character data, we cannot "see" the data in a binary file; it looks like hieroglyphics. If you have ever accidentally opened a binary file in a text editor, you have seen these strange results. The block read function is file read (*fread*). The block write function is file write (*fwrite*).

File Read: *fread*

The function *fread,* whose declaration is shown below, reads a specified number of bytes from a binary file and places them into memory at the specified location.

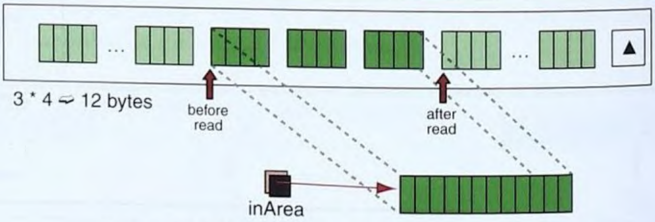
```
int fread (void* pInArea,
           int   elementSize,
           int   count,
           FILE* sp);
```

The first parameter, `pInArea`, is a pointer to the input area in memory. Note that a generic (void) pointer is used. This allows any pointer type to be passed to the function.

File read expects a pointer to the input area, which is usually a structure. This is because binary files are most often used to store structures.[1] However, C gives us the flexibility to read any type of data, from a character to a complex structure or even a multidimensional array.

The next two elements, `elementSize` and `count`, are multiplied to determine how much data are to be transferred. The size is normally specified using the *sizeof* operator and the count is normally one when reading structures.

The last parameter is the associated stream. Figure 13-7 is an example of a file read that reads data into an array of integers. When *fread* is called, it transfers the next three integers from the file to the array, `inArea`.



```
fread (inArea, sizeof (int), 3, spData);
```

FIGURE 13-7 File Read Operation

[1] In other programming languages, structures stored in files are known as records.

The code to read the file is shown in Program 13-1.

PROGRAM 13-1 Read File of Integers

```
// Read a file of integers, three integers at a time.
{
  …
// Local Declarations
  FILE* spIntFile;
  int   itemsRead;
  int   intAry[3];

// Statements
  spIntFile = fopen("int_file.dat", "rb");
  …
  while ((itemsRead = fread(intAry,
       sizeof(int), 3, spIntFile)) != 0)
    {
     // process array
        …
    } // while
  …
}  // block
```

File read returns the number of items read. In Figure 13-7, it will range between 0 and 3 because we are reading three integers at a time. For example, assume that when we try to read the file, only two integers are left to be read. In this case, *fread* will return 2. If we return and try to read the file again, *fread* will then return 0.

Note that *fread* does not return end of file—it returns the number of elements read. End of file is detected in the above situation when we called *fread* with fewer than three integers left in the file. The question is, how can we tell that we are at the end of file? C provides another input/output function, *feof,* to test for end of file. We discuss this function in "File Status Functions" later in this section.

Now let's look at a more common use of *fread:* reading structures (records). Assume that we have defined a structure that stores data about students. Given the type of data that will be stored, we would expect the structure to contain some string data and other data, such as integers or real numbers. One advantage of block input/output functions is that they can transfer these data one structure (record) at a time. A second advantage is that the data do not need to be formatted. Figure 13-8 shows the operation of *fread* when a structure is being read.

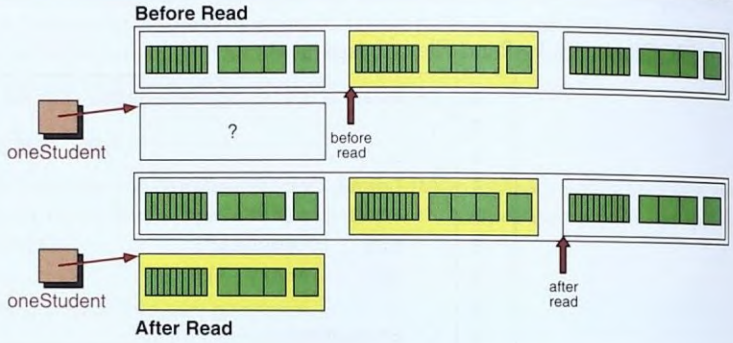


FIGURE 13-8 Reading a Structure

The code shown in Program 13-2 reads the file.

PROGRAM 13-2 Read Student File

```
/* Reads one student's data from a file
      Pre   spStuFile is opened for reading
      Post  stu data structure filled
            ioResults returned
*/
int readStudent (STU* oneStudent, FILE* spStuFile)
{
// Local Declarations
   int ioResults;

// Statements
   ioResults = fread(oneStudent,
                     sizeof(STU), 1, spStuFile);
   return ioResults;
} // readStudent
```

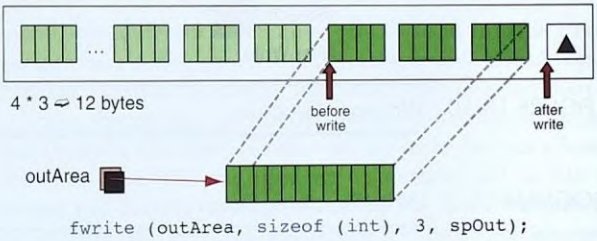
Program 13-2 Analysis Different companies have different standards. One company with which we are familiar has a standard that programs shall have only one read and one write statement for each file. The standard was created to make it easier to make changes to the programs. Program 13-2 is a typical implementation of this standard. One difficulty with this type of function, however, is that it is impossible to generalize the steps that are to be taken for various input results, such as error handling and end of file. Therefore, we pass the input/output result back to the calling function for analysis and action.

File Write: *fwrite*

The function *fwrite,* whose declaration is shown below, writes a specified number of items to a binary file.

```
int fwrite (void*  pOutArea,
            int    elementSize,
            int    count,
            FILE*  sp);
```

The parameters for file write correspond exactly to the parameters for the file read function.



```
fwrite (outArea, sizeof (int), 3, spOut);
```

FIGURE 13-9 File Write Operation

Functionally, *fwrite* copies `elementSize` x `count` bytes from the address specified by `pOutArea` to the file. It returns the number of items written. For example, if it writes three integers, it returns 3. We can use the return value, therefore, to test the write operation. If the number of items written is fewer than `count`, then an error has occurred. Depending on the device we are working with, it may be possible to repeat the write, but generally the program should be aborted when we get a write error. Figure 13-9 shows the write operation that parallels the read in Figure 13-7.

Assuming that we are writing a file of student structures, Figure 13-10 shows the write operation. Program 13-3 contains the code for the function that writes the data.

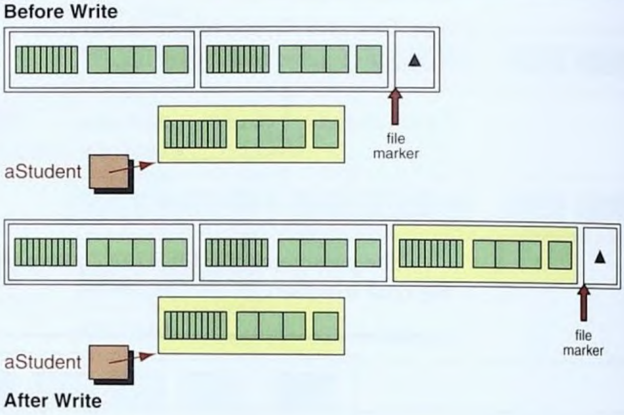


FIGURE 13-10 Writing a Structure

PROGRAM 13-3 Write Structured Data

```
/* Writes one student's record to a binary file.
      Pre  aStudent has been filled
           spOut is open for writing
      Post aStudent written to spOut
*/
void writeStudent (STU* aStudent, FILE* spOut)

{
// Local Declarations
   int ioResult;

// Statements
   ioResult = fwrite(aStudent,
                     sizeof(STU), 1, spOut);
   if (ioResult != 1)
      {
       printf("\a Error writing student file \a\n");
       exit (100);
      } // if
   return;
}  // writeStudent
```

Program 13-3 Analysis Contrast this write structured data function with the one that we wrote to read data. Although it is not possible to generalize on the action to be taken if data are not read, it is possible to do so with write errors. If the program cannot write data, it must be aborted. Therefore, we put the error checking and action in the write function itself.

File Status Functions

C provides three functions to handle file status questions: test end of file (*feof*), test error (*ferror*), and clear error (*clearerr*).[2]

Test End of File: *feof*

The *feof* function is used to check if the end of file has been reached. If the file is at the end—that is, if all data have been read— the function returns nonzero (true). If end of file has not been reached, zero (false) is returned. The function declaration is shown below.

```
int feof (FILE* stream);
```

In general, two different techniques can be used to detect end of file. Some languages have a look-ahead function. When look-ahead logic is being used, the system transfers the current data to the program and then reads the next data. Under this design, we can detect the end of file at the same time that we read (transfer data back to our work area) the last data from the file. The second technique, the one used by C, detects end of file when we attempt to read and there is nothing left on the file. Even if all the data have been read from the file, *feof* does not return true until we attempt to read beyond the last data.

Test Error: *ferror*

Test error (*ferror*) is used to check the error status of the file. Errors can be created for many reasons, ranging from bad physical media (disk or tape) to illogical operations, such as trying to read a file in the write state. The *ferror* function returns true (nonzero) if an error has occurred. It returns false (zero) if no error has occurred. The function declaration is shown below.

```
int ferror (FILE* stream);
```

Note, however, that testing for an error does not reset the error condition. Once a file enters the error state (see Figure 13-4) it can only return to a read or write state by calling clear error (see below).

Clear Error: *clearerr*

When an error occurs, the subsequent calls to *ferror* return nonzero, until the error status of the file is reset. The function *clearerr* is used for this purpose. Its function declaration is shown in the next example.

```
void clearerr (FILE* stream);
```

[2] Although the C99 Standard introduced the Boolean type, most standard functions continue to use the traditional nonzero for true and zero for false return status.

Note, however, that even though we have cleared the error, we have not necessarily cured the problem. We may find that the next read or write returns to the error state.

Positioning Functions

Positioning functions have two uses. First, for randomly processing data in disk files (we cannot process tape files randomly), we need to position the file to read the desired data. Second, we can use the positioning functions to change a file's state. Thus, if we have been writing a file, we can change to a read state after we use one of the positioning functions. It is not necessary to change states after positioning a file, but it is allowed.

We will discuss three file position functions: rewind, tell location, and file seek.

Rewind File: *rewind*

The *rewind* function simply sets the file position indicator to the beginning of the file (Figure 13-11). The function declaration is shown below.

```
void rewind(FILE* stream);
```

A common use of the rewind function is to change a work file from a write state to a read state. Often it is necessary to place data in a file temporarily for later processing. When all the data have been written and we are ready to begin reading, we rewind the file and simply start reading. Remember, however, that to read and write a file with only one open, we must open it in update mode, in this case, `w+` or `w+b`.

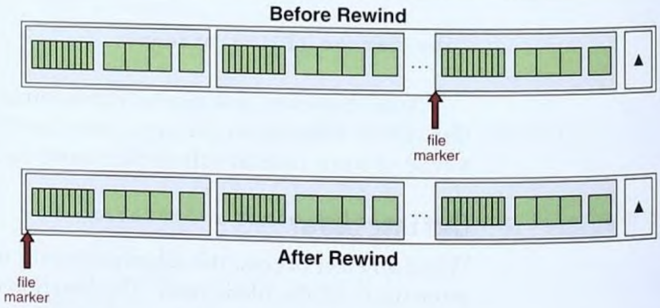


FIGURE 13-11 Rewind File

The same effect can be accomplished by closing the output file and opening it as input. However, the rewind is a faster operation.

Current Location: *ftell*

The *ftell* function reports the current position of the file marker in the file, relative to the beginning of the file. Recall that C considers files as streams of data. It measures the position in the file by the number of bytes, relative to zero, from the beginning of the file. Thus, when the file position indicator is at the beginning of the file, *ftell* returns zero. If the file position indicator is at the second byte of the file, *ftell* returns 1, representing the position 1 byte offset from the beginning of the file. The function declaration for *ftell* is shown below.

```
long int ftell (FILE* stream);
```

Note that *ftell* returns a long integer. This is necessary because many files have more than 32,767 bytes, which is the maximum integer value on many computers. The operation of *ftell* is graphically shown in Figure 13-12.

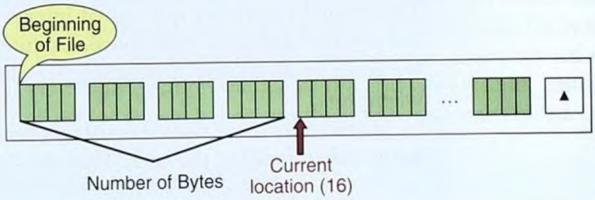


FIGURE 13-12 Current Location (*ftell*) Operation

Another important factor to consider is that *ftell* returns the number of bytes from the beginning of the file. This is true even when we read or write structures. If we need to know the structure number relative to the first structure, then we must calculate it. This can be done by dividing the *ftell* return value by the size of the structure, as shown below.

```
numChar   = ftell (sp);
numStruct = numChar / sizeof(STRUCTURE_TYPE);
```

In Figure 13-12, *ftell* returns 16. Since each structure is 4 bytes, the result of the calculation shown above is 4, which means that there are four structures before the current location. Another way to look at it is that the file is positioned at the fourth integer relative to zero.

If *ftell* encounters an error, it returns –1. We know of only two conditions that can cause an error. The first, using *ftell* with a device that cannot store data, such as the keyboard, is a program logic or design error. The second error occurs when the position is larger than can be represented in a long integer. Obviously, this could occur only with very large files, but files of more than a million records are common in industry.

The primary purpose of *ftell* is to provide a data address (offset) that can be used in a file seek. It is especially useful when we are dealing with text files for which we cannot calculate the position of data.

Set Position: *fseek*

The *fseek* function positions the file location indicator to a specified byte position in a file. It gets its name from the disk-positioning operation, seek. Seek moves the access arm on a disk to a position in the file for reading or writing. Since this is exactly the purpose of **seek file**, it is an appropriate name. Its function declaration is shown below.

```
int fseek(FILE* stream, long offset, int wherefrom);
```

The first parameter is a pointer to an open file. Since the seek is used with both reading and writing files, the file state can be either read or write. The second parameter is a signed integer that specifies the number of bytes the position indicator must move absolutely or relatively. To understand what we mean by absolutely or relatively, we must first discuss the third parameter, *wherefrom*.

C provides three named constants that can be used to specify the starting point (*wherefrom*) of the seek. They are shown below.

```
#define SEEK_SET 0
#define SEEK_CUR 1
#define SEEK_END 2
```

When *wherefrom* is `SEEK_SET` or `0`, then the offset is measured absolutely from the beginning of the file. This is the most common use of file seek. Thus, to set the file indicator to byte 100 on a file, we would code the following statement:

```
fseek(sp, 99L, SEEK_SET);
```

We can use zero in place of `SEEK_SET`. If you are puzzling over the second parameter in the above statement, remember that the file position is relative to zero and must be a long integer. Actually, the compiler is smart enough to convert an integer value to long integer, but it is more efficient if we specify the correct type, especially with literals.

Now let's look at the *wherefrom* option, `SEEK_CUR`. If *wherefrom* is `SEEK_CUR` or `1`, then the displacement is calculated relatively from the current file position. If the displacement is negative, the file's position moves back toward the beginning of the file. If it is positive, it moves forward toward the end of the file. It is an error to move beyond the beginning of the file. If we move beyond the end of the file, the file is extended, but the contents of the extended bytes are unknown. Whenever we extend the file, there is always the possibility of running out of space, which is an error. To position the file marker to the next record in a structured file, we execute the following statement:

```
fseek(sp, sizeof(STRUCTURE_TYPE), SEEK_CUR);
```

To position the student file described earlier at the structure indicated by the integer variable `stuLoc`, we use the following statement:

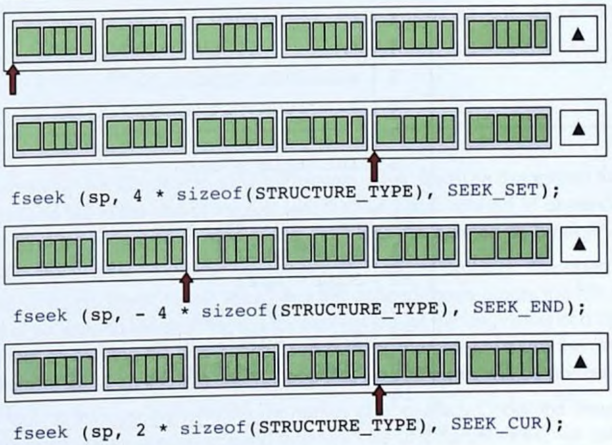
```
fseek(stuFile, (stuLoc - 1)* sizeof(STU), SEEK_SET);
```

It is necessary to adjust the integer location, `stuLoc`, by subtracting 1 to convert the ordinal structure number to a zero base. That is, if `stuLoc` contains 55, indicating we want to read the 55th student in the file, we position the file to the location of the 54th student relative to zero.

Finally, if *wherefrom* is `SEEK_END` or `2`, the file location indicator is positioned relative to the end of the file. If the offset is negative, the file position marker is moved backward toward the beginning of the file; if it is positive, it extends the file. This technique can be used to write a new record at the end of the file. We simply position the file at the end with a `SEEK_END` and a displacement of zero as shown below and then write the new record.

```
fseek(stuFile, 0L, SEEK_END);
```

The seek function returns zero if the positioning is successful. It returns nonzero if the positioning is not successful. Figure 13-13 shows the effect of *fseek* in different situations.



```
fseek (sp, 4 * sizeof(STRUCTURE_TYPE), SEEK_SET);
fseek (sp, - 4 * sizeof(STRUCTURE_TYPE), SEEK_END);
fseek (sp, 2 * sizeof(STRUCTURE_TYPE), SEEK_CUR);
```

FIGURE 13-13 File Seek Operation

The file seek is intended primarily for binary files. It does, however, have limited functionality with text files. We can position a text file to the beginning using *fseek* with a zero offset from the beginning of the file (SEEK_SET). However, *rewind* provides the same functionality and is more appropriate for text files. To position a text file at the end, we can use *fseek* with a zero offset and a *wherefrom* SEEK_END, as discussed earlier.

We cannot use file seek to position ourself in the middle of a text file unless we have used *ftell* to record the location. The reasons for this have to do with control codes, newlines, vertical tabs, and other nuisances of text files. However, if we have saved a location using *ftell* and want to go back to that position, we can use *fseek*, as shown below.

```
fseek(sp, ftell_location, SEEK_SET);
```

Note that since *ftell* returns a position relative to the beginning of the file, we must use SEEK_SET when we reposition the file.

EXAMPLE 13-1 Block Input/output: Append Files

Let's look at a program that reads and writes binary files. Suppose, for example, that we had two copies of files with integer data. Perhaps one file represents data from one week and the other file represents data for a second week. We want to combine both files into a single file. The most efficient way to do this is to append the data from one file to the end of the other file. This logic is shown in Program 13-4.

PROGRAM 13-4 Append Two Binary Files

```
/* This program appends two binary files of integers.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
// Local Declarations
FILE* sp1;
FILE* sp2;
int   data;
long  dataCount;
char  fileID[13];

```

continued

PROGRAM 13-4 Append Two Binary Files *(continued)*

```
17  // Statements
18     printf("This program appends two files.\n");
19     printf("Please enter file ID of the primary file: ");
20     scanf("%12s", fileID);
21     if (!(spl = fopen (fileID, "ab")))
22         printf("\aCan't open %s\n", fileID), exit (100);
23
24     if (!(dataCount = (ftell (spl))))
25        printf("\a%s has no data\n", fileID), exit (101);
26     dataCount /= sizeof(int);
27
28     printf("Please enter file ID of the second file: ");
29     scanf("%12s", fileID);
30     if (!(sp2 = fopen (fileID, "rb")))
31        printf("\aCan't open %s\n", fileID), exit (110);
32
33     while (fread (&data, sizeof(int), 1, sp2) == 1)
34        {
35         fwrite (&data, sizeof(int), 1, spl);
36         dataCount++;
37        } // while
38
39     if (! feof(sp2))
40        printf("\aRead Error. No output.\n"), exit (120);
41
42     fclose (spl);
43     fclose (sp2);
44
45     printf("Append complete: %ld records in file\n",
46             dataCount);
47     return 0;
48  }  // main
```

Program 13-4 Analysis First, note the way the files are opened in Program 13-4. Because it appends the data from the second file to the end of the first file, the first file is opened in append mode and the second file is opened in read mode. Both files are opened as binary.

Opening the first file in append mode presents a minor problem: The open is successful even if the file doesn't exist. Recall that the append mode places the file marker at the end of an existing file or, if there is no existing file, at the beginning of a new file. To ensure that an existing file was opened successfully, therefore, we use the *ftell* function in statement 24. If an existing file was opened, `dataCount` will be nonzero.

Now look at the way the external file names are handled. Under the assumption that this program will be used to append different files at different times, we asked the user to enter the file names. This technique provides maximum flexibility for generalized programs.

Statement 22 contains one of the few valid uses of multiple statements on one line—in this case, multiple expressions separated by the comma operator. Why is this statement considered valid when we have so strongly emphasized one statement per line? The answer is that we are handling an error situation that should never occur. In good human engineering style, we want to deemphasize the error logic so that it doesn't distract the reader. Therefore, we place the print error message and exit in one expression.

The heart of the program is contained in statements 33 through 37. As long as the read is successful, we keep going. When we reach the end of file, the read returns zero and the loop terminates. We now have another problem, however. We don't know if the read terminated at end of file or because of a read error. We therefore use the *feof* function to make sure we read to the end of the file.

The program concludes with a printed message that contains the number of records on the appended file. Since we didn't read all the data on the first file, however, we must calculate the number of integers in it. To do this, we used the result of the *ftell* that verified that the first file existed. Recall that *ftell* returns the number of bytes to the current location in the file—in this case, the end of the file. To get the number of integers, we simply divide by the size of one integer. (See statement 26.) Then, each time we write, we add 1 to the record count. This gives us the number of integers in the combined file.

System File Operations

A few functions operate on the whole file instead of the contents. These functions generally use operating system calls to perform operations such as remove a file, rename a file, or create a temporary binary file.

Remove File: *remove*

The *remove* function removes or deletes the file using its external name. The parameter is a pointer to the name of the file. Its function is shown below.

```
int remove (char* filename);
```

It returns zero if the deletion is successful. It returns nonzero if there is an error, such as the file can't be found. For example, if we want to delete a file named `file1.dat`, we execute the following statement.

```
if (remove ("file1.dat"))
    printf("Error, file cannot be deleted");
```

Any attempt to access a file after it has been removed will result in an error.

Rename File: *rename*

When we create a new version of a file and want to keep the same name, we need to rename the old version of the file. The *rename* function declaration is shown below.

```
int rename  (const char* oldFilename,
             const char* newFilename);
```

Both the old name and the new name must be given as parameters. The rename function returns zero if renaming is successful; it returns nonzero if there is an error.

For example, in a DOS system, if we want to rename a student file and designate it a backup, we could use the rename function as shown below.

```
if (rename ("STUFILE.DAT", "STUFILE.BAK"))
    printf("Error, the file cannot be renamed");
```

Create Temporary File: *tmpfile*

The *tmpfile* function creates a new temporary output file. Although we could do the same thing with an *fopen* in the w+b mode, the difference is that the file is available only while the program is running. It will be closed and erased when the execution of the program is finished. It is a temporary file, not a permanent one. Its function declaration is

```
FILE*  tmpfile (void);
```

To create a temporary file, we first define a file pointer and then open it, as shown below.

```
FILE*  sp;

sp = tmpfile ();
```

Now we can write to the file as we would to any file. Because the file is opened in the w+b mode, if we want to read from the file we must reposition it using one of the repositioning functions such as *rewind* or *fseek*.

13.3 Converting File Type

A rather common but somewhat trivial problem is to convert a text file to a binary file and vice versa. C has no standard functions for these tasks. We must write a program to make the conversion. We describe the file conversion logic in this section.

Creating a Binary File from a Text File

To create a binary file, we usually start with data provided by the user. Since the user is providing the data, it will be in human readable form—that is, in text form. If only a small amount of initial data are required, they are often read from a keyboard. With a lot of data, however, it is easier for the user to enter the data with a text editor and then read the text file and create the binary file.

When we read the text file, we can use either the *fscanf* function to convert the data as they are being read, or the *fgets* and *sscanf* functions to read the data a line at a time and then convert the data to the proper internal format. As the data are being converted, they are placed in a structure. At the end of each line, the structure is written to the binary file. This process is repeated until the text file has been completely converted to the binary structure. The structure chart for this program is shown in Figure 13-14.

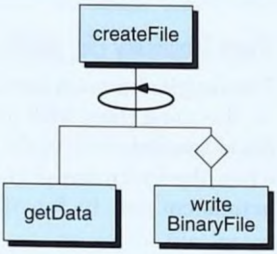


FIGURE 13-14 Create Binary File Structure Chart

Let's assume that we want to convert student data to a binary file. The data consist of a student's name, ID, three exams, eight problems, and a final grade. In the text file version, each field is separated by one or more whitespace characters and each student's data are stored on a separate line. The create file program is shown in Program 13-5.

PROGRAM 13-5 Text to Binary Student File

```
/* Reads text file of student data & creates binary file.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// Type Declarations
```

continued

PROGRAM 13-5 Text to Binary Student File *(continued)*

```
typedef struct stuData
   {
    char  name[26];
    char  id[5];
    int   exams[3];
    int   problems[8];
    char  grade;
   } STU_DATA;

// Function Declarations
bool getData         (FILE*     textFile,
                      STU_DATA* aStudent);
void writeBinaryFile (STU_DATA* aStudent,
                      FILE*     binFile);

int main (void)
{
// Local Declarations
   char* textFileID = "P13-stu.txt";
   char* binFileID  = "P13-stu.bin";

   STU_DATA aStudent;

   FILE* textFile;
   FILE* binFile;

// Statements
   printf("\nBegin Student Binary File Creation\n ");
   if (!(textFile = fopen(textFileID, "r")))
      {
       printf("\nCannot open %s\n", textFileID);
       exit (100);
      } // if textFile
   if (!(binFile = fopen(binFileID, "wb")))
      {
       printf("\nCannot open %s\n", binFileID);
       exit (200);
      } // if binFile

   while (getData (textFile, &aStudent))
      writeBinaryFile (&aStudent, binFile);

   fclose(textFile);
   fclose(binFile);
```

continued

PROGRAM 13-5 Text to Binary Student File *(continued)*

```
   printf("\n\nFile creation complete\n");
   return 0;
}  // main

/* ==================== getData ====================
   This function reads the text file.
      Pre   textFile is opened for reading
      Post  data read and returned
*/
bool getData (FILE* textFile, STU_DATA* aStu)
{
// Local Declarations
   char  buffer[100];

// Statements
   fgets(buffer, sizeof(buffer), textFile);
   if (!feof(textFile))
      {
      sscanf(buffer, "%s %s %d%d%d%d%d%d%d%d%d%d%d %c",
         aStu->name, aStu->id,
        &aStu->exams[0],
        &aStu->exams[1],&aStu->exams[2],
        &aStu->problems[0], &aStu->problems[1],
        &aStu->problems[2], &aStu->problems[3],
        &aStu->problems[4], &aStu->problems[5],
        &aStu->problems[6], &aStu->problems[7],
        &aStu->grade);
       return true;
      } // if
   return false;
} // getData

/* ================= writeBinaryFile =================
   Write student data to a binary file.
      Pre   binFile is opened as a binary output file
            aStudent is complete
      Post  Record written
*/
void writeBinaryFile (STU_DATA* aStudent,
                      FILE* binFile)
{
// Local Declarations
   int amtWritten;

```

continued

PROGRAM 13-5 Text to Binary Student File *(continued)*

```
// Statements
   amtWritten = fwrite (aStudent,
                        sizeof(STU_DATA), 1, binFile);
   if (amtWritten != 1)
      {
       printf("Can't write student file. Exiting\n");
       exit (201);
      } // if
   return;
} // writeBinaryFile
```

Program 13-5 Analysis

Note how we specified the external file names as strings. This makes it easy to change the file names when necessary. It also allows us to identify the file by name if the open fails.

The program starts and ends with a message that identifies what program is running and that it has successfully completed. This is a good programming technique and a standard in many organizations.

The *while* loop is controlled by the results of the getData function call. To understand how it works, therefore, first look at getData. When you need to read a lot of text data, especially if it is coming from a file, the preferred technique is to use the get string (*fgets*) function and then use the string scan (*sscanf*) function to convert it into internal binary formats. The get string function reads text data until it finds a newline character, an end of file, or the maximum number of characters has been read. It returns the address of the input string, in this case buffer. When end of file is detected, it returns a null pointer.

To make this function robust, we would break the *sscanf* into several calls, one for the first two strings, one for the exams, one for the problems, and one for the grade at the end. This would allow us to easily verify that the correct data were read by examining the returned value from *sscanf*.

Now that we understand that the getData function returns either an address or *NULL*, we understand how the *while* statement in *main* works. It simply loops until an end of file is detected—that is, until *NULL* is returned.

Creating a Text File from a Binary File

Programmers convert a binary file to a text file in two situations. The first is when we need to display the data for people to read; we discuss this below. The second is when it is necessary to export the data to another system, which can't read the binary file. This occurs, for example, if the word sizes for integers and floats are different on the two different hardware systems. As long as all lines are formatted the same and they use the same character alphabet, text files are portable.

An interesting problem is to create a report of the data in the binary file. Obviously, we need to read the binary file and write the data as a text file, but there is much more to it than that. First, the report needs a name, so each page must have a title. The title should include the report date and a page number. To make the report meaningful, each column should have a column caption. Finally, the report should have an end of report message as the last line of the report so that the user knows that all data have been reported.

Since we put a title on the top of each page, we need to know when a page is full. This is generally done by counting the number of lines on a page and when the count exceeds the maximum, skipping to the next page. Good structured programming requires that the heading logic be in a separate function. The design for printing the student data is shown in Figure 13-15.

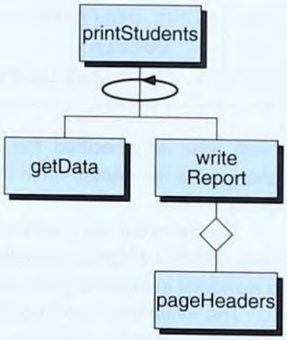


FIGURE 13-15 Design for Print Student Data

We write the data to a file so that it can be sent to the printer when we want a hard copy. You should be aware of one final point before looking at the program. In the print functions, we use a type modifier known as the *static* storage class (see statements 97 and 128). We discuss the concept of storage classes in Appendix K. For now, you only need to know that *static* keeps the contents of a variable available between calls. The first time we call a function with a static variable, the variable is initialized. After that, its value is retained between calls. This is a great C feature that reduces the need to pass parameters just to retain a local variable. The program is shown in Program 13-6.

PROGRAM 13-6 Print Student Data

```
 1  /* Reads a binary file of student data, and prints it.
 2         Written by:
 3         Date:
 4  */
 5  #include <stdio.h>
 6  #include <stdlib.h>
 7
 8  #define MAX_LINES_PER_PAGE  50
 9  #define BUFFER_SIZE        133
10  #define FORM_FEED          '\f'
11
```

continued

PROGRAM 13-6 Print Student Data *(continued)*

```
// Type Declarations
typedef struct stuData
  {
   char  name[26];
   char  id[5];
   int   exams[3];
   int   problems[8];
   char  grade;
  } STU_DATA;

// Function Declarations
STU_DATA getData      (FILE* binFile);
void     writeReport  (STU_DATA aStudent,
                       FILE* prtFile);
void     pageHeaders  (FILE* prtFile);

int main (void)
{
// Local Declarations
   char      stuFileID[]  = "P13-stu.bin";
   char      prtFileID[]  = "P13-stu.prt";
   STU_DATA  aStudent;
   FILE*     stuFile;
   FILE*     prtFile;

// Statements
   printf("\nBegin Student Report Creation\n ");

   if(!(stuFile = fopen(stuFileID, "rb")))
      {
       printf("\nCannot open %s\n", stuFileID);
       exit (100);
      } // if stuFile
   if (!(prtFile = fopen(prtFileID, "w")))
      {
       printf("\nCannot open %s\n", prtFileID);
       exit (200);
      } // if prtFile

   aStudent = getData (stuFile);
   while (!feof(stuFile))
      {
       writeReport (aStudent, prtFile);
       aStudent = getData (stuFile);
```

continued

PROGRAM 13-6 Print Student Data *(continued)*

```
      } // while

   fprintf(prtFile, "\nEnd of Report\n");
   fclose(stuFile);
   fclose(prtFile);
   printf("\n\nEnd Student Report Creation\n");
   return 0;
}  // main

/* =================== getData ===================
   This function reads the student binary file.
      Pre   stuFile is opened for reading
      Post  one student record read and returned
*/
STU_DATA getData (FILE* stuFile)
{
// Local Declarations
   int       ioResult;
   STU_DATA  aStu;

// Statements
   ioResult = fread(&aStu,
                    sizeof(STU_DATA), 1, stuFile);
   if (!ioResult)
      if (!feof(stuFile))
         {
          printf("\n\nError reading student file\n");
          exit (100);
         } // if !feof
   return aStu;
} // getData

/* ================== writeReport ==================
   Write student report to a text file.
      Pre   prtFile is opened as a text output file
            aStudent is complete
      Post  Report line written
*/
void writeReport (STU_DATA  aStu, FILE* prtFile)
{
// Local Declarations
   static int lineCount = MAX_LINES_PER_PAGE + 1;
   char      buffer[BUFFER_SIZE];

```

continued

PROGRAM 13-6 Print Student Data *(continued)*

```
// Statements
   if (++lineCount > MAX_LINES_PER_PAGE)
   {
     pageHeaders  (prtFile);
     lineCount = 1;
    } // if

   sprintf (buffer,
     "%-25s %4s %4d%4d%4d%4d%4d%4d%4d%4d%4d%4d%4d %c\n",
      aStu.name, aStu.id,
      aStu.exams[0],    aStu.exams[1],     aStu.exams[2],
      aStu.problems[0],
      aStu.problems[1], aStu.problems[2],
      aStu.problems[3], aStu.problems[4],
aStu.problems[5],
      aStu.problems[6], aStu.problems[7],
      aStu.grade);
   fputs (buffer, prtFile);
   return;
} // writeReport

/* ================== pageHeaders ==================
   Writes the page headers for the student report.
      Pre   prtFile is opened as a text output file
      Post  Report headers and captions written
*/
void pageHeaders (FILE* prtFile)
{
// Local Declarations
   static int pageNo = 0;

// Statements
   pageNo++;
   fprintf(prtFile, "%c", FORM_FEED);
   fprintf(prtFile, "%-66s  Page %4d\n",
         "Student Report ", pageNo);
   fprintf(prtFile, "%-25s %-6s %-10s %-27s Grade\n\n",
         "Student Name", "ID", "Exams", "Problems");

   return;
}  // pageHeaders
// ================= End of Program =================
```

Program 13-6 Analysis Even though this program is rather simple, you should note the following points:

First, we have declared the maximum number of lines per page and the print buffer size as preprocessor-defined constants. This makes it easy to change them should it be necessary. It also makes it easy to set the print logic so that it will print the header the first time through the function. This leads us to the second point you should note. The logic for `pageHeaders` will cause the first page to be blank. That is, we issue a page form feed before any data have been written. This is standard in production programs, but you may want to change it so that the first page is not wasted. In this case, you will have to call the `pageHeaders` function before you start the file reading to write the first headings and move the form feed write to just before statement 103.

The file is read using look-ahead logic. At statement 51, we read the first record of the file. This allows us to use the *feof* function to control the *while* loop. At the end of the loop, we read the file again. Although this loop closely resembles a posttest loop, there is one difference. If the file is empty, a post-test loop would fail. By reading the first record from the file before the loop, we ensure that we can process any file condition using the simplest possible logic.

Note the way we handle the report title and line captions in `pageHeaders`. Many programmers simply try to code them in the format string. This works, but it takes a lot of trial and error to get it right. Our technique simply adds the widths from the data write and uses them for the widths in the caption prints. The program may still need a little manual adjustment, but it is a much simpler approach.

Finally, note how we formatted the student output. We used *sprintf* to format a line of output. There is no significant advantage to using the string format function rather than the print function, but we wanted to demonstrate its use. After the data have been formatted, we use *fputs* to write them. Note also the way we aligned the data for readability. Since it takes several lines, we group the common data on lines by themselves. This makes it much easier to read the code. We could have used *for* statements to print the array data, but that would have been less efficient.

13.4 File Program Examples

This section contains two common file applications. The first uses the file positioning functions to randomly process the data in a file. The second merges two files.

EXAMPLE 13-2 Random File Accessing

Program 13-7 demonstrates the concept of randomly accessing data in a file. We begin by creating a binary file of integers. Each integer is the square of the data's position in the file, relative to 1. After the file has been created, we print it in sequence, starting at the beginning of the file. We then print it in a random sequence using *fseek* and a random number generator.

PROGRAM 13-7 Random File Application

```
/* Shows application of some functions we have studied
   in this chapter. The program first creates a binary
   file of integers. It then prints the file, first
   sequentially and then randomly using rand().
      Written by:
      Date:
```

continued

PROGRAM 13-7 Random File Application *(continued)*

```
*/
#include <stdio.h>
#include <stdlib.h>

// Function Declarations
void buildFile   (FILE** sp);
void printFile   (FILE*  sp);
void randomPrint (FILE*  sp);

int main (void)
{
// Local Declarations
   FILE* fpData;

// Statements
   buildFile   (&fpData);
   printFile   (fpData);
   randomPrint (fpData);
   return 0;
}  // main
```

Program 13-7 Analysis The *main* function simply calls three functions in order. The first function receives a pointer to the file pointer, which is the only variable declared in *main*. It is necessary to use double dereferencing here because `buildFile` needs to pass back the file pointer to *main*. The other two calls do not change the file pointer, they just use it. Therefore, it can be passed to them by value.

EXAMPLE 13-3 Build Random File

Program 13-8 builds a binary file.

PROGRAM 13-8 Random File: Build File

```
/* =================== buildFile ===================
   Creates a disk file that we can process randomly.
      Pre   nothing
      Post  file has been built
*/
void buildFile (FILE** spData)
{
// Local Declarations
   int  data;

// Statements
```

continued

PROGRAM 13-8 Random File: Build File *(continued)*

```
   if (!(*spData = fopen("SAMPLE.DAT", "w+b")))
      {
       printf("\aError opening file.\n");
       exit (100);
      } // if open
   for (int i = 1; i <= 10; i++)
       {
        data = i * i;
        fwrite(&data, sizeof(int), 1, *spData);
       } // for
   return;
}  // buildFile
```

Program 13-8 Analysis The `buildFile` function in Program 13-8 simply creates a file with ten records. Each record consists of a single integer, which is the square of the ordinal record number relative to 1 (not 0). The file is opened with write plus so that we can first write to it and then later in the program read it. Note that all references to the file use the dereference operator to update the file pointer, which exists in *main*.

EXAMPLE 13-4 Sequentially Print a Random File

Program 13-9 prints the file sequentially.

PROGRAM 13-9 Random File: Sequential Print

```
/* =================== printFile ===================
   Prints the file starting at the first record.
      Pre   spData is an open file
      Post  file has been printed
*/
void printFile (FILE* spData)
{
// Local Declarations
   int data;
   int recNum;

// Statements
   recNum = 0;
   rewind(spData);
   fread(&data , sizeof(int), 1, spData);
   while (!feof(spData))
      {
       printf("Record %2d: %3d\n", recNum++, data);
       fread(&data, sizeof(int), 1, spData);
```

continued

PROGRAM 13-9 Random File: Sequential Print *(continued)*

```
      } // while
   return;
} // printFile
```

```
Results:
Record 0:   1
Record 1:   4
Record 2:   9
Record 3:  16
Record 4:  25
Record 5:  36
Record 6:  49
Record 7:  64
Record 8:  81
Record 9: 100
```

Program 13-9 Analysis The `printFile` function in Program 13-9 simply reads the file sequentially starting at the beginning (Record 0). Study the *while* loop. Note that we have coded it with the first read before the loop. We use this technique so that we can detect end of file in the *while* statement. The *while* loop prints the current record and then reads the next record. When all records have been processed, the read will detect end of file, and the loop will terminate with all records processed.

EXAMPLE 13-5 Randomly Print Random File

Program 13-10 prints the file randomly.

PROGRAM 13-10 Random File: Random Print

```
/* =================== randomPrint ===================
   This function randomly prints the file. Some data
   may be printed twice, depending on the random
   numbers generated.
      Pre  spData is an open file
      Post Ten records have been printed
*/
void randomPrint (FILE* spData)
{
// Local Declarations
   int data;
   int randomSeek;

// Statements
```

continued

PROGRAM 13-10 Random File: Random Print *(continued)*

```
   printf("\nFile contents in random sequence.\n");
   for (int i = 0; i < 10; i++)
      {
       randomSeek =  (rand() % 10);
       fseek(spData,
             sizeof(int) * randomSeek, SEEK_SET);
       fread(&data, sizeof(int), 1, spData);
       printf("Record %3d ==> %3d\n",
               randomSeek, data);
      } // for
   return;
} // randomPrint
```

```
Results:
File contents in random sequence.
Record   8 ==>  81
Record   8 ==>  81
Record   3 ==>  16
Record   5 ==>  36
Record   1 ==>   4
Record   7 ==>  64
Record   0 ==>   1
Record   9 ==> 100
Record   2 ==>   9
Record   6 ==>  49
```

Program 13-10 Analysis The `randomPrint` function is the most interesting of these functions. We use a *for* loop to print ten records. Within the loop, we use a random number generator to determine which record we will read next. Since there are ten records in the file, we set the random number to modulo ten, which gives us potential record numbers from zero to nine. This corresponds exactly with the file on disk, which occupies record positions zero to nine. The function output created using our computer is shown in Program 13-10. Note that it starts by printing Record 8 twice and then prints all other records other than Record 4. Compare the output to the file print in Program 13-9.

Merge Files

In "Block Input/Output Functions," in Section 13.2, we discussed the concept of combining two files by appending the data. In this section, we discuss another way to combine data: the file merge. When we **merge** data from two files, the result is one file with the data ordered in key sequence. This requires that we completely read two input files and create a new output file. This concept is shown in Figure 13-16.

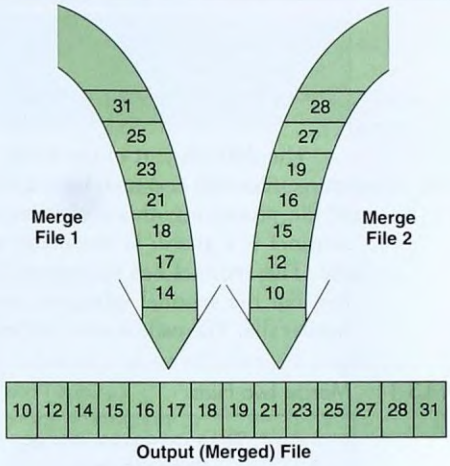


FIGURE 13-16 File Merge Concept

The merge files pseudocode is shown in Algorithm 13-1. The design is rather simple. We start by reading data from each merge file. We then (1) compare the data in the two files, (2) write the smaller to the merge output file, (3) read the next record from the file whose record was written, and (4) continue the loop.

ALGORITHM 13-1 Pseudocode for Merging Two Files

```
Algorithm MergeTwoFiles
This program merges two files
 1 input (File1, Rec1)
 2 input (File2, Rec2)
 3 HighSentinel = high-value
 4 loop (not eof(File1)) OR (not eof(File2))
   1  if  Rec1.Key  <=  Rec2.Key  then
      1  output (File3, Rec1)
      2  input  (File1, Rec1)
      3  if eof(File1)
         1  Rec1.Key = HighSentinel
      4  end if
   2  else
      1  output (File3, Rec2)
      2  input  (File2, Rec2)
      3  if eof(File2)
         1  Rec2.Key = HighSentinel
      4  end if
```

continued

ALGORITHM 13-1 Pseudocode for Merging Two Files *(continued)*

```
   3  end if
 5 end loop
End MergeTwoFiles
```

The difficult part of the merge design is the end-of-file logic. One of the merge files will end first, but we never know which one. To simplify the end of file processing, this design introduces a concept known as a sentinel. A sentinel is a guard; in our merge algorithm, the sentinel guards the end of file. The sentinel has the property that its value is larger than any possible key. For the sentinel value, we use `INT_MAX`, which is found in the *limits.h* header file. The code is seen in Program 13-11.

PROGRAM 13-11 Merge Two Files

```c
/* This program merges two files
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define READ_MODE  "rb"
#define WRITE_MODE "wb"

int main (void)
{
// Local Declarations
   FILE*  spM1;
   FILE*  spM2;
   FILE*  spOut;

   int  recM1;
   int  recM2;
   int  sentinel    = INT_MAX;
   int  mergeCnt    = 0;

   char file1ID[]   = "P13Mrg1.bin";
   char file2ID[]   = "P13Mrg2.bin";
   char fileOutID[] = "P13Mrg3.bin";

// Statements
   printf("Begin File Merge:\n");
   if (!(spM1 = fopen (file1ID, READ_MODE)))
```

continued

PROGRAM 13-11 Merge Two Files *(continued)*

```
       printf("\aError on %s\n", file1ID), exit (100);

   if (!(spM2 = fopen (file2ID, READ_MODE)))
       printf("\aError on %s\n", file2ID), exit (200);

   if (!(spOut = fopen (fileOutID, WRITE_MODE)))
      printf("\aError on %s\n", fileOutID), exit (300);

   fread (&recM1, sizeof(int), 1, spM1);
   if (feof(spM1))
        recM1 = sentinel;
   fread (&recM2, sizeof(int), 1, spM2);
   if (feof(spM2))
       recM2 = sentinel;

   while (!feof(spM1) || !feof(spM2))
      {
       if (recM1 <=  recM2)
          {
           fwrite (&recM1, sizeof(int), 1, spOut);
           mergeCnt++;
           fread  (&recM1, sizeof(int), 1, spM1);
           if (feof(spM1))
              recM1 = sentinel;
          } // if
       else
          {
           fwrite (&recM2, sizeof(int), 1, spOut);
           mergeCnt++;
           fread  (&recM2, sizeof(int), 1, spM2);
           if (feof(spM2))
              recM2 = sentinel;
          } // else
      } // while
   fclose (spM1);
   fclose (spM2);
   fclose (spOut);
   printf("End File Merge. %d items merged.\n",
           mergeCnt);
   return 0;
}  // main
```

Program 13-11 Analysis We have written this simple program without any subfunctions. Once again we have used the multiple-statement error message format for the error message and exit after each open statement.

Programs that involve the comparison of data from two files require that the first record from both files be read before any comparisons can be made. This is sometimes called "priming the files." The reads that prime the files are coded before the main *while* loop. Since duplicate read statements are required in the loop, you might wonder why we didn't use a *do...while*. The reason is that the program would fail if both files were empty. As we coded it, the program works if either or both merge files are empty.

Study the logic at statements 54 and 62 carefully. These statements implement the sentinel concept. When a file reaches its end, we set the key in the record area for the file to the sentinel value. This ensures that all the data on the other file will compare low and be written to the output file.

The most difficult statement in this simple program is the *while* statement. We need to keep looping as long as either of the files is not at the end of file. The most straightforward code is as we coded it in statement 46. Another common way to code it is shown below.

```
(!(feof(spM1) && feof(spM2)))
```

Use De Morgan's (see Chapter 5) rule to prove to yourself that these two statements are identical.

13.5 Software Engineering

Any file environment requires some means of keeping the file current. Data are not static; they are constantly changing, and these changes need to be reflected in their files. The function that keeps files current is known as **updating**. To complete our discussion of files, we discuss some of the software engineering design considerations for file updating. For this discussion, we assume a student binary file similar to the ones we have discussed in the chapter.

Update Files

Three specific files are associated with an update program. First is the permanent data file, called the master file. The **master file** contains the most current computer data for an application.

The second file is the **transaction file**. It contains changes to be applied to the master file. Three basic types of changes occur in all file updates; others may also occur, depending on the application. *Add transactions* contain data about new students to be added in the master file; *delete transactions* identify students who will be deleted from the file; and *change transactions* contain revisions to specific student records in the file.

To process any of these transactions, we need a key. A **key** is one or more fields that uniquely identify the data in the file. For example, in the student file, the key would be student ID. In an employee file, the key would be Social Security number.

The third file needed in an update program is an **error report file**. An update process is very rarely error free. When an error occurs, we need to report it to the user. The *error report* contains a listing of all errors discovered during the update process and is presented to the user for corrective action.

File updates are of two types: batch and online. In a **batch update**, changes are collected over time and then all changes are applied to the file at once. In an **online update**, the user is directly connected to the computer and the changes are processed one at a time—often as the change occurs.

Sequential File Update

For our discussion, we will assume a batch, sequential file environment. A **sequential file** is a file that must be processed serially starting at the beginning. It does not have any random processing capabilities. The sequential master file has the additional attribute that it is ordered on the key.

A sequential file update actually has two copies of the master file, the old master and the new master. This is because whenever a sequential file is changed, it must be entirely re-created—even if only one student's score on one exam is changed.

Figure 13-17 contains an environment chart for a sequential file update. In this chart, we see the four files we discussed above. We use the tape symbol for the files because it is the classic symbol for sequential files. Sequential files could just as easily be stored on a disk. Note that after the update program completes, the new master file is sent to off-line storage, where it is kept until it is needed again. When the file is to be updated, the master file is retrieved from the off-line storage and used as the old master.

Generally, at least three copies of a master file are retained in off-line storage, in case it becomes necessary to regenerate an unreadable file. This retention cycle is known as the grandparent system because three generations of the file are always available: the grandparent, the parent, and the child.

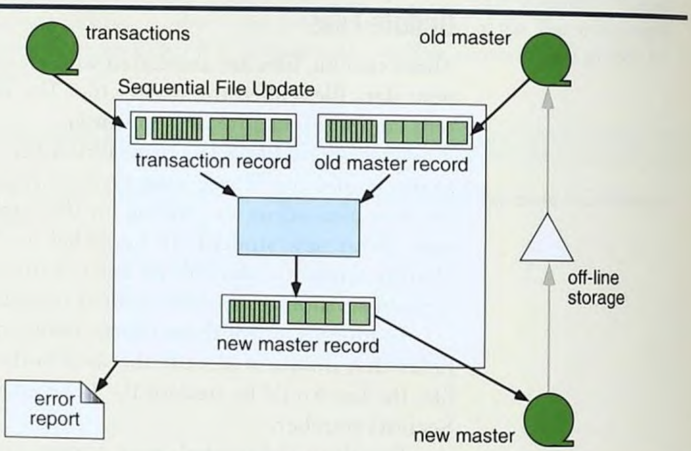


FIGURE 13-17 Sequential File Update Environment

The Update Program Design

In 1981, a computer scientist named Barry Dwyer published an update algorithm in the *Communications of the ACM* that was so elegant that it has become a classic.[3] We have adapted his algorithm for our discussion.

Since a sequential master file is ordered on a key and since it must be processed serially, the transaction file must also be ordered on the same key. The update process requires that we match the keys on the transaction and master file and, assuming that there are no errors, take one of the following three actions:

1. If the transaction file key is less than the master file key, add the transaction to the new master.

[3] Barry Dwyer, "One More Time—How to Update a Master File," *Communications of the ACM*, 24, no. 1 (January 1981): 3–8.

2. If the transaction file key is equal to the master file key, either
    a. Change the contents of the master file data if the transaction is a revise transaction, or
    b. Remove the data from the master file if the transaction is a delete.
3. If the transaction file key is greater than the master file key, write the old master file record to the new master file.

This updating process is shown in Figure 13-18. In the transaction file, the transaction codes are `A` for add, `D` for delete, and `R` for revise. The process begins by matching the keys for the first record on each file, in this case,

```
14 > 10
```

Thus, Rule 3 is used, and we write the master record to the new master record. We then match 14 and 13, which results in 13 being written to the new master. In the next match, we have

```
14 == 14
```

Thus, according to Rule 2a, we use the data in the transaction file to change the data in the master file. However, we do not write the new master file at this time. More transactions may match the master file, and we need to process them, too.

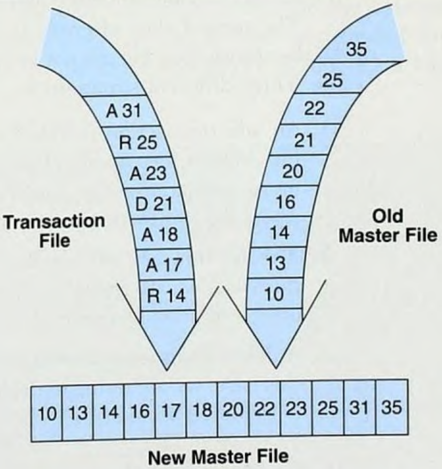


FIGURE 13-18 File Updating Example

After writing 16 to the new master, we have the following situation:

```
17 < 20
```

According to Rule 1, we must add 17 to the new master file. We do this by copying the transaction to the new master file, but again, we don't write it yet. This newly added record may have some revision transactions, and we need to be able to process them. For example, this capability is needed when a new student registers and adds classes on the same day. The computer has to be able to add the new student and then process the class registrations in the same batch run. We write the new master for 17 when we read transaction 18.

The processing continues until we read the delete transaction, at which time we have the following situation:

```
21 == 21
```

and since the transaction is a delete, according to Rule 2b, we need to drop 21 from the new master file. To do this, we simply read the next master record and transaction record without writing the new master. The processing continues in a similar fashion until all records on both files have been processed.

Update Errors

Two general classes of errors can occur in an update program. The user can submit bad data, such as a grade that is not A, B, C, D, or F. For our discussion, we will assume that no data errors are present. Detecting data errors is the subject of data validation and has been discussed.

The second class of errors is file errors. File errors occur when the data on the transaction file are not in synchronization with the data on the master file. Three different situations can occur:

1. An add transaction matches a record with the same key on the master file. Master files do not allow duplicate data to be present. When the key on an add transaction matches a key on the master file, therefore, we reject the transaction as invalid and report it on the error report.
2. A revise transaction's key does not match a record on the master file. In this case, we are trying to change data that do not exist. This is also a file error and must be reported on the error report.
3. A delete transaction's key does not match a record on the master file. In this case, we are trying to delete data that do not exist, and this must also be reported as an error.

Update Structure Chart

The structure chart for the sequential file update is shown in Figure 13-19. In this structure chart, `process` contains the updating function.

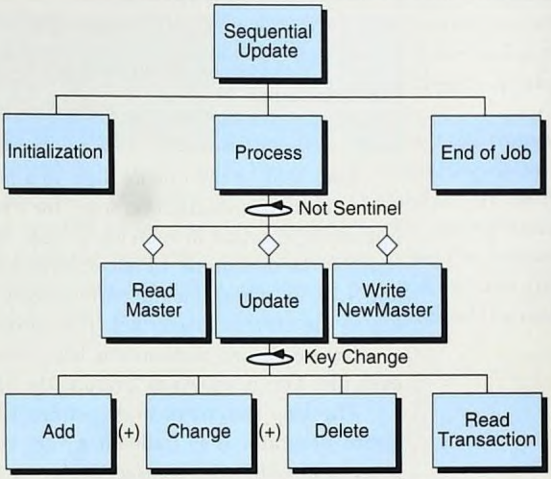


FIGURE 13-19 Update Structure Chart

Update Logic

`Initialization` is a function that opens the files and otherwise prepares the environment for processing. `EndofJob` is a function that closes the files and displays any end of job messages. The mainline processing is done in `Process`.

Although it is beyond our scope to develop the complete set of update functions, it is important that you at least understand the mainline logic found in `Process`. Its pseudocode is shown in Algorithm 13-2.

ALGORITHM 13-2 Pseudocode for File Update

```
Algorithm Sequential Update
 1 read first record from transaction file
 2 read first record from old master file
 3 select next entity to be processed
 4 loop current entity not sentinel
   1 if current entity equals old master entity
     1 copy old master to new master work area
     2 read old master file
   2 end if
   3 if current entity equals transaction entity
     1 update new master work area
   4 end if
   5 if current entity equals new master entity
     1 write new master file
```

continued

ALGORITHM 13-2 Pseudocode for File Update

```
   6  end if
   7  select next entity to be processed
 5 end loop
End Sequential Update
```

Let's look at the update logic in a little more detail. The first three statements contain initialization logic for Process. The driving force behind the update logic is that in each *while* loop, we process all the data for one student (entity). To determine which student we need to process next (statements 3 and 4.7), we determine the current entry by comparing the current transaction key to the current master key. The current key is the smaller.

Before we can compare the keys, however, we must read the first record in each file. This is known as "priming the files" and is seen in statements 1 and 2.

The *loop* statement in Algorithm 13-2 contains the driving logic for the entire program. It is built on a very simple principle: As long as data are present in either the transaction file or the master file, we continue to loop. When a file has been completely read, we set its key to a sentinel value. When both files are at their end, therefore, both of their keys will be sentinels. Then, when we select the next student to be processed, it will be a sentinel, which is the event that terminates the *while* loop.

Three major processing functions take place in the *while* loop. First, we determine if the student on the old master file needs to be processed. If it does, we move it to the new master output area and read the next student from the old master file. The key on the old master can match the current key in two situations: a change or delete transaction exists for the current student. This logic is seen in statement 4.1.

The second major process handles transactions that match the current student. It calls a function that determines the type of transaction being processed (add, change, or delete) and handles it accordingly. If it is an add, it moves the new student's data to the new master area. If it is a change, it updates the data in the new master area. And if it is a delete, it clears the key in the new master area so that the record will not be written. To handle multiple transactions in the update function, it reads the next transaction and continues if its key matches the current student.

The last major process writes the new master when appropriate. If the current student matches the key in the new master file area, then the record needs to be written to the file. This will be the case unless a delete transaction was processed.

Summary

In this section we looked at a very important algorithm, the classic sequential file update, and discussed its mainline logic flow.

The elegance of Dwyer's algorithm, as seen in Algorithm 13-2, lies in the determination of the current student and the separation of the update process into three distinct functions: read the old master, update the current student, and write the new master. Study the concept of the current student carefully, and make sure you understand how it controls the three major processes in the loop. Then, with a little thought, you should be able to develop the other functions in the update program.

Also note that Algorithm 13-2 contains only one read transaction function. Its function is to read a valid transaction. Therefore, it contains all of the simple data validation logic to determine if the data are correct. It does not perform any file errors; they are handled in the update function. If any errors are found, the read transaction function writes the transaction to an error report and reads the next transaction. When it returns to the calling function, it has either read a valid transaction or has found the end of the file.

13.6 Tips and Common Programming Errors

Refer to the "Tips and Common Programming Errors" section in Chapter 7. Many of those tips apply to binary files as well.

1. `EOF` is type integer, and its value is normally –1. Therefore, if you want to test the value of a variable to `EOF`, use an integer variable, not a character. In most systems, a character variable cannot store a negative number (–1 here).
2. Remember to open a file before using it.
3. You can create a file for writing; but to read a file, it must exist.
4. When you open a file for writing using `w` mode, you must close it and open it for reading (`r` mode) if you want to read from it. To avoid this problem, you can open it in `w+` mode.
5. An open file can be in one of the three states: read, write, or error. If you want to switch from read to write or from write to read, you must use one of the file-positioning functions.
6. Do not open a file in `w` mode when you want to preserve the contents of the file. Opening a file in `w` mode erases the contents of the file.
7. Remember that in general you cannot print the contents of a binary file. It must be converted to a text file first.
8. Unlike other input/output functions, the first parameter of the *fread* and *fwrite* functions is a pointer to the input area, not a file pointer. The file parameter is the last (fourth) parameter.
9. The second parameter of the *fread* and *fwrite* functions is the size of the element, and the third parameter is the count of elements.
10. The *fread* and *fwrite* functions return the number of elements read or written, not the number of bytes read or written.
11. Remember that *feof* does not look ahead. It returns true only if an attempt is made to read the end of file.
12. The second parameter in the *fseek* function is the number of bytes, not the number of elements.
13. When the third parameter in the *fseek* function is `SEEK_END (2)`, the middle parameter should normally be a negative long integer to access a byte in the file. If it is positive, it is referring to a byte after the end of file.
14. To add an element at the end of the file, you use the *fseek* function with the second parameter set to zero and the third parameter to the value of `SEEK_END`.
15. Remember that every time you use the *fread* or *fwrite* function, it automatically advances the file position indicator toward the end of the file the number of bytes equal to the size of the element times the number of elements read or written.
16. It is good practice to close all files before terminating a program.

17. It is a compile error to refer to a file with its external file name rather than its file pointer.

18. It is a compile error to omit the file pointer when using a file function. This error most often occurs when using the file versions of the file format functions, such as *fprintf*.

19. It is a logic error to refer to a file before it is opened.

20. It is a logic error to open a file for reading when it doesn't exist. This is usually an error in the external file name in the open statement.

21. It is a logic error to attempt to read from a file in the write state and vice versa. This is true even when the file has been opened in the update mode.

22. Opening an output disk file will fail if the disk is full.

23. Opening an existing file in write mode deletes the file. If your input file disappears, check your open modes.

24. It is a logic error to use *fseek* to place the file marker before the first byte of a file.

13.7 Key Terms

batch update
end of file
error report file
error state
key
master file
merge
online update
read state
seek file
sequential file
transaction file
update mode
write state

13.8 Summary

- ❑ A file is a collection of related data stored in an auxiliary storage device.
- ❑ A stored stream of 0s and 1s can be interpreted as either a text or a binary file.
- ❑ A text file is a file of characters.
- ❑ A binary file is a collection of related data stored in the internal format of the computer.
- ❑ A file is always in one of the following states: read, write, or error.
- ❑ A binary file can be opened in one of the following modes: `rb`, `r+b`, `wb`, `w+b`, `ab`, and `a+b`.
- ❑ The *fread* function reads the number of bytes specified by the product of the element size and the number of elements.

- ❑ The *fwrite* function writes the number of bytes specified by the product of the element size and the number of elements.
- ❑ The *feof* function checks for end of file.
- ❑ The *ferror* function is used to check the error status of a file.
- ❑ The *clearerr* function is used to clear an error.
- ❑ The *rewind* function sets the file position indicator to the beginning of the file.
- ❑ The *ftell* function tells you the current position of the file position indicator.
- ❑ The *fseek* function positions the file position indicator to the beginning of a specified byte.
- ❑ The *remove* function removes or deletes a file from the disk.
- ❑ The *rename* function renames a file on the disk.
- ❑ The *tmpfile* function creates a new temporary file.
- ❑ To be kept current, master files must be updated.
- ❑ The master file update requires four files: the old master, a new master, a transaction file, and an error report file.
- ❑ The basic master file transactions are add, revise, and delete.

13.9 Practice Sets

Review Questions

1. A file can be read only if it is in the read state.

   a. True
   b. False

2. A file opened in the `r+b` mode is a binary file opened in the read state and update mode.

   a. True
   b. False

3. Binary files are read using the block input/output function *fread*.

   a. True
   b. False

4. Using *fseek* to position a file beyond the current end of file places the file in an error state.

   a. True
   b. False

5. To merge two files, you can use the merge function found in the standard input/output library.

   a. True
   b. False

6. Which of the following statements about files is true?

   a. All files must be opened before they can be used.
   b. Binary files are more portable than text files.
   c. Binary files are slower than text files.
   d. Files must be closed.
   e. Text files contain data stored in the internal format of the computer.

7. Which of the following statements about files is false?

   a. Because they are more flexible, binary files are more portable.
   b. Binary files contain data stored in the internal format of the computer.
   c. Binary files usually contain records in the form of structured data.
   d. Text files store only character data.
   e. Some text file data need to be converted to internal storage formats for processing.

8. The ________________ results when a failure occurs during an open or during either a read or write operation.

   a. error state
   b. fail state
   c. read state
   d. update state
   e. write state

9. If a file in the read state is written to, then the following occurs:

   a. The file is placed in an error state regardless of the file mode.
   b. The file is placed in an error state unless it was opened in the update write mode (`w+b`).
   c. The file state is automatically switched if the file is opened for updating.
   d. The program is aborted regardless of how the file was opened.
   e. The program is aborted unless the file was opened for updating.

10. Which of the following file modes opens a file in the write state for updating?

    a. `ab`
    b. `a+b`
    c. `rb`
    d. `r+b`
    e. `wb`

11. Which of the following C functions is used to output data to a binary file?

    a. *fwrite*
    b. *output*

c. *write*
d. *writef*
e. *fprintf*

12. Which of the following is a file status function?

a. *feof*
b. *freport*
c. *frewind*
d. *fseek*
e. *ftell*

13. The ______________ function may be used to position a file at the beginning.

a. *fclose*
b. *feof*
c. *freport*
d. *fseek*
e. *ftell*

14. A ______________ is one or more fields that uniquely identify the data in a file.

a. field
b. identifier
c. key
d. structure
e. transaction

15. Which of the following statements about sequential file updating is false?

a. Add, change, and delete transactions are used to update the file.
b. Sequential files are often updated in an online environment.
c. Sequential files contain structures (records) with a key to identify the data.
d. The file must be processed starting at the beginning.
e. The file must be entirely re-created when it is updated.

Exercises

16. Explain the difference between the following pairs of modes:

```
rb and r+b
wb and w+b
ab and a+b
```

17. Find the error(s) in the following code. (Assume the PAY_REC type has been properly declared.)

```
char*   m = "wb";
char*   str = "Payroll";
PAY_REC payRec;
FILE*   sp;
sp = fopen (str, m);
fread(payRec, sizeof(payRec), 1, sp);
```

18. Given the following declarations and assuming that the file is open:

```
FILE* sp;
char  s[20];
```

find any errors in each of the following statements:

a. `fread(s, 20, sp);`
b. `fread(s, 20, 1, sp);`
c. `fread(s, 1, 20, sp);`
d. `fread(sp, 1, 20, s);`
e. `fread(sp, 20, 1, s);`

19. Given the following declarations and assuming that the file is open:

```
FILE* sp;
char  s[20];
```

find any errors in each of the following lines:

a. `locn = ftell(sp);`
b. `locn = ftell(1, sp);`
c. `fseek(0, 20L, sp);`
d. `fseek(sp, 20L, 0);`
e. `fseek(sp, 20L, 1);`

20. What would be printed from the following program? Draw a picture of the file with the file marker to explain your answer.

```
#include <stdio.h>
int main (void)
{
 long int pos;
 FILE*    sp;

 sp = fopen("SAMPLE.DAT", "w+b");
 for (char c = 'A'; c <= 'E'; c++)
      fwrite(&c , sizeof(char), 1, sp);
 pos = ftell (sp);
 printf("The position of the file marker is : %ld",
        pos);
 return 0;
} // main
```

21. What would be printed from the following program? Draw a picture of the file with the file marker to explain your answer.

```
#include <stdio.h>
int main (void)
{
 char  c;
 FILE* sp;

 sp = fopen("SAMPLE.DAT", "w+b");
 for (c = 'A'; c <= 'E'; c++)
      fwrite(&c, sizeof(char), 1, sp);
 fseek(sp, 2, 0);
 fread(&c, 1, 1, sp);
 printf("\n\n%c", c);
 return 0;
} // main
```

22. What would be printed from the following program? Draw a picture of the file with the file marker to explain your answer.

```
#include <stdio.h>
int main (void)
{
 char  c;
 FILE* sp;

 sp = fopen("SAMPLE.DAT", "w+b");
 for (c = 'A'; c <= 'E'; c++)
      fwrite(&c, sizeof (char), 1, sp);
 rewind(sp);
 fread(&c, 1, 1, sp);
 printf("\n\n%c", c);
 return 0;
} // main
```

23. What would be printed from the following program? Draw a picture of the file with the file marker to explain your answer.

```
#include <stdio.h>
int main (void)
{
 char  c;
 long  int pos;
 FILE* sp;

 sp = fopen("SAMPLE.DAT", "w+b");
 for (c = 'A'; c <= 'E'; c++)
      fwrite(&c, sizeof(char), 1, sp);
```

continued

```
 pos = ftell (sp);
 pos--;
 pos--;
 fseek(sp, pos, 0);
 fread(&c, 1, 1, sp);
 printf("\n\n%c", c);
 return 0;
} // main
```

24. What would be printed from the following program? Draw a picture of the file with the file marker to explain your answer.

```
#include <stdio.h>
int main (void)
{
 char  c;
 long  int pos;
 FILE* sp;

 sp = fopen("SAMPLE.DAT", "w+b");
 for (c = 'A'; c <= 'E'; c++)
      fwrite(&c , sizeof(char), 1, sp);
 pos = ftell(sp);
 pos--;
 pos--;
 fseek(sp, -pos, 1);
 fread(&c, 1, 1, sp);
 printf("\n\n%c", c);
 return 0;
} // main
```

25. What would be printed from the following program? Draw a picture of the file with the file marker to explain your answer.

```
#include <stdio.h>
int main (void)
{
 char     d;
 long int pos;
 FILE*    sp;

 sp = fopen("SAMPLE.DAT", "w+b");
 for (char c = 'A'; c <= 'E'; c++)
      fwrite(&c , sizeof(char), 1, sp);
 pos = ftell(sp);
 pos--;
 pos--;
 fseek(sp, 2, 2);
```

continued

```
 fseek(sp, 1, 1);
 fread(&d, 1, 1, sp);
 printf("\n\n%c", d);
 return 0;
} // main
```

26. What would be printed from the following program? Draw a picture of the file with the file marker to explain your answer.

```
#include <stdio.h>
int main (void)
{
 long int pos;
 FILE*    sp;

 sp = fopen("SAMPLE.DAT", "w+b");
 for (int i = 1; i <= 5; i++)
      fwrite(&i , sizeof(int), 1, sp);
 pos = ftell(sp);
 printf("The position of the file marker is : %ld",
         pos);
 return 0;
} // main
```

27. What would be printed from the following program? Draw a picture of the file with the file marker to explain your answer.

```
#include <stdio.h>
int main (void)
{
 int   i;
 FILE* sp;

 sp = fopen("SAMPLE.DAT", "w+b");
 for (i = 1; i <= 5; i++)
      fwrite(&i , sizeof(int), 1, sp);
 fseek(sp, sizeof (int) * 2, 0);
 fread(&i, sizeof (int), 1, sp);
 printf("%d\n", i);
 return 0;
} // main
```

28. What would be printed from the following program? Draw a picture of the file with the file marker to explain your answer.

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

continued

```
 int   i;
 FILE* sp;

 sp = fopen ("SAMPLE.DAT", "w+b");
 for (i = 1; i <= 5; i++)
      fwrite(&i, sizeof(int), 1, sp);
 fseek(sp, -sizeof(int) * 2, 1);
 fread(&i,  sizeof(int), 1, sp);
 printf("%d", i);
 fclose (sp);
 return 0;
} // main
```

29. What would be printed from the following program? Draw a picture of the file with the file marker to explain your answer. Is the answer strange? Why?

```
#include <stdio.h>
int main (void)
{
 int   i;
 FILE* sp;

 sp = fopen("SAMPLE.DAT", "w+b");
 for (i = 1; i <= 5; i++)
      fwrite(&i, sizeof(int), 1, sp);
 fseek(sp, 7, 0);
 fread(&i, sizeof(int), 1, sp);
 printf("%d", i);
 fclose (sp);
 return 0;
} // main
```

30. What would be printed from the following program? Draw a picture of the file with the file marker to explain your answer.

```
#include <stdio.h>
int main (void)
{
 int      i;
 long int pos;
 FILE*    sp;

 sp = fopen("SAMPLE.DAT", "w+b");
 for (i = 1; i <= 5; i++)
      fwrite(&i, sizeof(int), 1, sp);
 pos = ftell(sp);
 pos -= 2 * sizeof(int);
```

continued

```
    fseek(sp, pos, 0);
    fread(&i, sizeof(int), 1, sp);
    printf("%d", i);
    fclose (sp);
    return 0;
   } // main
```

Problems

31. Write a function that copies the contents of a binary file of integers to a second file. The function must accept two file pointers and return an integer (zero representing a processing error and nonzero indicating successful completion).

32. Write a function that prints a specified number of records from the beginning of a file. The function is to accept two parameters. The first is a pointer to a binary file of structure type STR. The second is an integer that specifies the number of records to be printed (inclusive). The structure type is shown below:

```
typedef struct
   {
    int   i;
    float f;
   } STR;
```

If any errors occur, such as fewer records in the file than specified, it should return zero. Otherwise, it returns nonzero.

33. Write a function that compares two files and returns equal or not equal based on the result of the comparison. The functions should receive file pointers to two opened files and compare them byte by byte.

34. Write a function that returns the number of items in a binary file.

35. Write a function that prints the last integer in a binary file of integers.

36. Write a function that physically removes all items with a specified value (data) from a binary file of structure STR. You may use a temporary file. The file may contain more than one record with the delete value. The key value removed is to be entered from the keyboard.

```
typedef struct
{
 int   data;
 char  c;
} STR;
```

37. Write a function that appends one binary file at the end of another. Use the structure described in Problem 36.

38. Write a function that, given a binary file, copies the odd items (items 1, 3, 5, ..., *n*) to a second binary file and the even items (items 2, 4, 6, ..., *n*) to a third binary file. After all items have been copied, print the contents of both output files.

39. Write a function that reads items from a binary file and copies them to a dynamically allocated array. The function must first find the size of the binary file to allocate the array.

40. Write a function that takes a binary file of long integers and appends a new long integer at the end that is the sum of all integers in the original file.

Projects

41. A company has two small warehouses. The list of the products in each warehouse is kept in a text file (`InvFile1` and `InvFile2`), with each line representing information about one product. The manager wants to have only one list showing information about all products in both warehouses. Therefore, the two text files must be combined into one single text file (`OutFile`).

    Write a program that will copy information from the two text files (`InvFile1` and `InvFile2`) to two binary files (`BinFile1` and `BinFile2`). After creating the binary files, the program is to merge the two binary files to produce a combined binary file. After the combined binary file has been created, create a report file that can be printed. The report file is to contain page headers with an appropriate title and page numbers. The structure for the files is shown below.

```
typedef struct inv_rec
   {
    char  partNo[5];
    char  partName[15];
    int   qtyOnHand;
   } INV_REC;
```

42. A company keeps a list of parts that it purchases, with a line of information for each part that gives the part's unique code, name, and three codes for three suppliers that supply that particular part. This list is kept in a binary file and is sorted in ascending order according to the supplier's code.

    The company also keeps a list of its suppliers, with a line of information for each supplier, which gives the supplier's unique code, name, and address. This list is also kept in a binary file, which is sorted in ascending order according to the supplier's code.

    Write a program that enables the user to enter a part's unique code and to receive a list of three suppliers. If the code is found, the program prints the names and addresses of the three suppliers. If the code is not found, it prints a message to tell the user that the code is not in the file. After each inquiry, the program gives the user the option to quit.

Each record in the part file is made up of a part's code, name, and the codes for three suppliers. The part's code is an integer; the name is a string with a maximum length of ten characters; and each supplier's code is an integer. Note that not all parts have three suppliers. For parts with less than three suppliers, a special supplier code of 0000 is used to indicate no supplier.

Each record in the supplier file is made up of a supplier's code, name, and address. The supplier's code is an integer, the name is a string with a maximum length of ten characters, and the address has a maximum length of 20 characters.

The output is to be formatted with the first line showing the data for the part and the following lines showing data for the suppliers, indented one tab.

Sample data for the files are shown in Tables 13-2 and 13-3. You will first need to write a file conversion program to create the binary files. We suggest that you create a text file version of each file with your text editor and then read it to create the binary version.

Part Code	Part Name	Supplier 1	Supplier 2	Supplier 3
1000	Pen	5010	5007	5012
1001	Pencil	5006	5008	0000
1002	Paper	5001	5000	5003
1003	Ball Pen	5013	5009	5014
1004	Folder	5009	5007	5002
1005	Pointer	5012	5006	5005
1006	Mouse	5012	0000	0000
1007	Monitor	5000	5002	5007

TABLE 13-2 Project 42 Part File

Supplier Code	Supplier Name	Supplier Address
5000	John Marcus	2322 Glen Place
5001	Steve Chu	1435 Main Ave.
5002	David White	2345 Steve Drive
5003	Bryan Walljasper	780 Rose Mary Street

continued

TABLE 13-3 Project 42 Supplier File

Supplier Code	Supplier Name	Supplier Address
5004	Andrew Tse	P. O. Box 7600
5005	Joanne Brown	1411 Donnybrook Square
5006	Lucy Nguyen	2345 Saint Mary Road
5007	Fred West	11 Duarte Rd.
5008	Dennis Andrews	14 California Ave.
5009	Leo Washington	134234 San Rosa Place
5010	Frankie South	12234 North Justin St.
5011	Amanda Trapp	1345 South Bush Circle
5012	Dave Lightfoot	222 George Territory Drive
5013	Danna Mayor	11 George Bush Street
5014	Robert Hurley	14 Republican Alley

TABLE 13-3 Project 42 Supplier File *(continued)*

To read a record on the files, you will need to determine its position on the file. This is easily done by subtracting 1000 from the `PartCode` for the part file and 5000 from the `SupplierCode` for the supplier file. Then use *fseek* to position the file for reading.

43. Write a program that builds a histogram (see Chapter 8) using a randomly accessed disk file. The program is to begin by creating a new file of 20 integers. Each integer will represent the accumulator for its relative position; the first for the number 0, the second for the number 1, and so forth until the last for the number 19. The program is to then use the random number generator to create 100 random numbers in the range 0...19. As each random number is generated, it will be displayed in a 10 × 10 matrix (10 lines of 10 numbers each) and 1 added to its accumulator on the disk using the *fseek* function. After the random numbers have been generated, the file is to be read sequentially and a histogram displayed.

44. Your stockbroker has an online inquiry system that allows you to check the price of stocks using your personal computer. Simulate this system as described below.

Each stock is assigned a unique integral number in the range 1000...5000. They are stored on the disk so that stock 1000 is stored in location 0, stock 1001 in location 1, stock 2010 in location 1010, and so forth. To calculate the disk address for a requested stock, your program subtracts 1000 from the stock number and uses the result as the address in the file. (This is actually a simplified version of a concept known as "hashing," which you will learn when you study data structures.)

The data for each stock are described as follows:

```
   structure
      stock key                short integer
      stock name               string[21]
      stock symbol             string[6]
      current price            floating-point number
      YTD High                 floating-point number
      YTD Low                  floating-point number
      Price-Earning Ratio      short integer
  (YTD: Year to Date)
```

Using data from your local newspaper, create a binary file of at least 20 stocks. Then write a menu-driven system that allows the user to request data on any individual stock.

In addition, provide a capability to get a report of up to 20 stocks at one time. When this option is requested, open a temporary work file, and write the requested stocks to the file. After the last stock has been entered, read the file (without closing it), and prepare the report.

Chapter 14
Bitwise Operators

While the preceding chapters contain the material required for a traditional first course in programming, we have not discussed all the basic capabilities of the C language. This chapter discusses the C bitwise operators.

The C language is well suited to system programming because it contains operators that can manipulate data at the bit level. For example, the Internet requires that bits be manipulated to create addresses for subnets.

C has two categories of **bitwise operators** that operate on data at the bit level: logical bitwise operators and shift bitwise operators. Before we discuss these operators, however, we need to introduce exact-size integer types.

To demonstrate these concepts we develop three programs. The first discusses a checksum calculation and develops an example that could be used to validate Internet transmissions. The second uses masks to determine the first or the last address in a block of Internet addresses. The third demonstrates a bit manipulation technique that can be used in polynomial calculations.

In Software Engineering, we present a case study to demonstrate an approach to designing structure charts.

Objectives

- ❑ To be able to use the bitwise logical operators in programs
- ❑ To be able to use the bitwise shift operators in programs
- ❑ To understand how to create and use masks to manipulate bits

14.1 Exact Size Integer Types

The integer types, such as *int* and *long*, discussed in Chapter 2, are machine dependent. In one computer, the size of *int* may be four bytes; in another computer it may be two bytes. While many bitwise applications work well on machine-dependent integer types, other applications need to assure that the size is fixed. For example, to manipulate an Internet address, we need to define an unsigned integer of size 32 bits (4 bytes). Beginning with C99, C allows us to define integer types of sizes 8, 16, 32, and 64 bits. They are defined in the *stdint.h* header file. Table 14-1 documents these types. As you will see, most of the time we use unsigned integers.

	Type	Description
Signed	`int8_t`	8-bit signed integer
	`int16_t`	16-bit signed integer
	`int32_t`	32-bit signed integer
	`int64_t`	64-bit signed integer
Unsigned	`uint8_t`	8-bit unsigned integer
	`uint16_t`	16-bit unsigned integer
	`uint32_t`	32-bit unsigned integer
	`uint64_t`	64-bit unsigned integer

TABLE 14-1 Fixed-size Integer Types

14.2 Logical Bitwise Operators

The logical operators look at data as individual bits to be manipulated. Four operators are provided to manipulate bits: bitwise *and* (`&`), bitwise inclusive *or* (`|`), bitwise exclusive *or* (`^`), and *one's complement* (`~`). The first three are binary operators; the one's complement is a unary operator.

Bitwise *and* Operator

The **bitwise *and*** (`&`—precedence 8) is a binary operator that requires two integral operands (character or integer). It does a bit-by-bit comparison between the two operands. The result of the comparison is 1 only when both bits are 1; it is 0 otherwise. Table 14-2 shows the result of bit-by-bit comparison.

First Operand Bit	Second Operand Bit	Result
0	0	0
0	1	0
1	0	0
1	1	1

TABLE 14-2 *And* Truth Table

Program 14-1 demonstrates the bitwise *and* operator.

PROGRAM 14-1 Simple Bitwise *And* Demonstration

```
/* Demonstrate bitwise AND operator
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

int main (void)
{
// Local Declarations
   uint16_t  num1 = 0x0257;
   uint16_t  num2 = 0xA463;
   uint16_t  res;

// Statements
   res = num1 & num2;

   printf ("Input and results in hexadecimal:\n");
   printf ("num1:   %#06X\n", num1);
   printf ("num2:   %#06X\n", num2);
   printf ("result: %#06X\n", res);

   return 0;
}  // main
```

```
Results:
Input and results in hexadecimal:
num1:   0X0257
num2:   0XA463
result: 0X0043
```

Program 14-1 Analysis The result of an *and* operation is logically the intersection of the two values. We can verify the result by manually calculating it using binary values.

Bitwise Inclusive *or* Operator

The **bitwise inclusive *or*** (|—precedence 6) is a binary operator that requires two integral operands (character or integer). It does a bit-by-bit comparison between the two operands. The result of the comparison is 0 if both operands are 0; it is 1 otherwise. Table 14-3 shows the result of bit-by-bit comparison.

First Bit	Second Bit	Result
0	0	0
0	1	1
1	0	1
1	1	1

TABLE 14-3 Inclusive *Or* Truth Table

Program 14-2 demonstrates the basic operation of the inclusive *or*.

PROGRAM 14-2 Simple Inclusive *or* Demonstration

```
/* Demonstrate the inclusive OR operator
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

int main (void)
{
// Local Declarations
   uint16_t  num1 = 0x0257;
   uint16_t  num2 = 0xA463;
   uint16_t  res;

// Statements
   res = num1 | num2;
   printf ("Input and results in hexadecimal:\n");
   printf ("num1: %#06X\n", num1);
   printf ("num2: %#06X\n", num2);
   printf ("res:  %#06X\n", res);
```

continued

PROGRAM 14-2 Simple Inclusive *or* Demonstration *(continued)*

```

   return 0;
}  // main
```

```
Results:
Input and results in hexadecimal:
num1: 0X0257
num2: 0XA463
res:  0XA677
```

Bitwise Exclusive *or* Operator

The **bitwise exclusive *or*** (^—precedence 7) is a binary operator that requires two integral operands. It does a bit-by-bit comparison between the two operands. The result of the comparison is 1 only if one of the operands is 1 and the other is 0; it is 0 if both operands' bits are 0 or 1—that is, if they are both the same. Table 14-4 shows the result of bit-by-bit comparison.

First Bit	Second Bit	Result
0	0	0
0	1	1
1	0	1
1	1	0

TABLE 14-4 Exclusive *Or* Truth Table

Program 14-3 demonstrates the basic operation of the bitwise exclusive *or.*

PROGRAM 14-3 Simple Exclusive *or* Demonstration

```
/* This program demonstrates the use of the exclusive or
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

int main (void)
{
```

continued

PROGRAM 14-3 Simple Exclusive *or* Demonstration *(continued)*

```
// Local Declarations
   uint16_t  num1 = 0x0257;
   uint16_t  num2 = 0xA463;
   uint16_t  res;

// Statements
   res = num1 ^ num2;

// Print results in hexadecimal
   printf ("Input and results in hexadecimal:\n");
   printf ("num1: %#06X\n", num1);
   printf ("num2: %#06X\n", num2);
   printf ("res:  %#06X\n", res);

   return 0;
}  // main
```

```
Results:
Input and results in hexadecimal:
num1: 0X0257
num2: 0XA463
res:  0XA634
```

One's Complement Operator

The **one's complement** (~—precedence 15) is a unary operator applied to an integral value (character or integer). It complements the bits in the operand; that is, it reverses the bit value. The result is 1 when the original bit is 0; it is 0 when the original bit is 1. Table 14-5 shows the result of the one's complement.

Original Bit	Result
0	1
1	0

TABLE 14-5 One's Complement Truth Table

Program 14-4 demonstrates the basic operation of the one's complement.

PROGRAM 14-4 One's Complement

```
/* Demonstrate use of one's complement
      Written by:
```

continued

PROGRAM 14-4 One's Complement *(continued)*

```
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

int main (void)
{
// Local Declarations
   uint16_t  num = 0x0257;
   uint16_t  res;

// Statements
   res = ~num;

// Print results in hexadecimal
   printf ("Input and results in hexadecimal:\n");
   printf ("num: %#06X\n", num);
   printf ("res: %#06X\n", res);

   return 0;
}  // main
```

```
Results:
Input and results in hexadecimal:
num: 0X0257
res: 0XFDA8
```

Checksum

One bitwise manipulation application is the calculation of a checksum. A checksum is a mathematical calculation used for error detection. For example, credit cards use a checksum to verify that the card number is valid. Another example involves the detection of a bad transmission through a communication channel. When the data are sent, a checksum is calculated and attached to the transmission. When the data are received, the checksum that includes the sender's checksum is recalculated. If the newly calculated checksum is not zero, an error has occurred.

The traditional Internet checksum uses a 16-bit calculation. To calculate the checksum, the following steps are used.

1. Set the checksum to 0.
2. Add 16-bit sections of data to the checksum using one's complement arithmetic. (See Appendix D.)
3. Complement the result.

We complement the result at the end because a number (`T`) and its complement (`-T`), when added always give `-0` in one's complement.

Figure 14-1 shows the calculation of a checksum for a small string, "ABCDEFGHI." Note that because the number of characters is nine, we include the null character (`\0`) at the end of the string to make five 16-bit values. The string is logically divided into groups of 16 bits (two characters), which are then added together using one's complement.

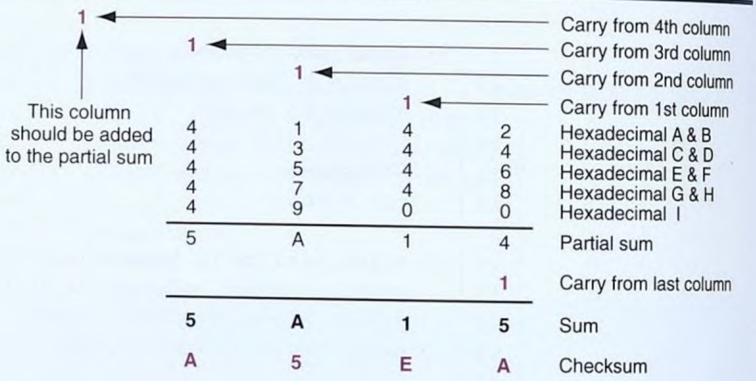


FIGURE 14-1 Checksum Calculation

Program 14-5 demonstrates the calculation of a checksum for the sender site. To properly weigh the first character, we multiply it by 256. To get the one's complement, we add the carry bits after the sum is calculated as shown in Figure 14-1. This is done using a loop. The loop extracts the higher (left) 16 bits and adds them to the lower (right) 16 bits.

PROGRAM 14-5 Demonstrate Checksum

```
/* Demonstrate the calculation of a checksum using one's
  complement arithmetic.
    Written by:
    Date:
*/
#include <stdio.h>
#include <string.h>
#include <stdint.h>

int main (void)
{
```

continued

PROGRAM 14-5 Demonstrate Checksum *(continued)*

```
// Local Declarations
   uint32_t sum      = 0x00000000;
   uint16_t checksum = 0x0000;
   char*    str      = "ABCDEFGHI";
   int      len;

// Statements
   len = strlen (str);
   if (len % 2 == 1)
      // Make the number of characters even
      len++;

   for (int i = 0; i < len; i += 2)
       sum = (sum + str[i] * 256 + str[i + 1]);

   // Add carries into lower 16 bits
   while (sum >> 16)
      sum = (sum & 0xffff) + (sum >> 16);

   // Complement
   checksum = ~sum;

   printf ("str:      %s\n",    str);
   printf ("checksum: %#06X\n", checksum);

   return 0;
}  // main
```

```
Results:
str:      ABCDEFGHI
checksum: 0XA5EA
```

14.3 Shift Operators

The shift operators move bits to the right or the left. When applied to unsigned numbers, these operators are implementation independent. When used with signed numbers, however, the implementation is left to the discretion of the software engineer who designs the compiler. It is often predicated on the hardware for which the compiler is being written. Therefore, they must be used with caution with signed numbers. Because the C standard leaves the implementation up to the compiler writer (there is no standard) *code that shifts signed negative numbers may not be portable to other platforms.*

Bitwise Shift-Right Operator

The **bitwise shift right** (>>—precedence 11) is a binary operator that requires two integral operands (character or integer). The first operand is the value to be shifted. The second operand specifies the number of bits to be shifted.

Shifting binary numbers is just like shifting decimal numbers. When bits are shifted right, the bits at the rightmost end are deleted. What is shifted in on the left, however, depends on the type and the implementation. If the type is unsigned, then the standard calls for zero bits to be shifted in. If the type is signed, however, the implementation may either shift in zeros or copies of the leftmost bit. Since the implementation is the system programmer's responsibility, any function that shifts signed negative values may not be portable. The shift-right operation is diagrammed in Figure 14-2.

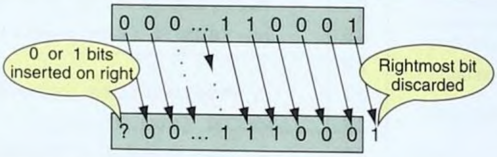


FIGURE 14-2 Shift-right Operation

Program 14-6 demonstrates the shift-right operation. Because C does not provide any formatted conversion for binary numbers, we wrote a simple function to print a 16-bit fixed integer variable as a binary number.

PROGRAM 14-6 Simple Shift-right Demonstration

```
/* Demonstrate the bitwise shift-right operator.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

// Function Declaration
void bin16 (uint16_t num, char* bitStr);

int main (void)
{
// Local Definitions
   uint16_t num  = 0x0040;
   uint16_t res;
```

continued

PROGRAM 14-6 Simple Shift-right Demonstration *(continued)*

```
   char     bitStr[17] = {0};

// Statements
   bin16 (num, bitStr);
   printf("Original value:  %s (%#06x)\n",
          bitStr, num);

   res = num >> 1;
   bin16 (res, bitStr);
   printf("Shifted 1 right: %s (%#06x)\n",
          bitStr, res);

   res =  num >> 2;
   bin16 (res, bitStr);
   printf("Shifted 2 right: %s (%#06x)\n",
          bitStr, res);

   res = num >> 4;
   bin16 (res, bitStr);
   printf("Shifted 4 right: %s (%#06x)\n",
          bitStr, res);

   return 0;
}  // main

/* ======================= bin16 ======================
   Convert fixed 16-bit integer to binary digit string.
      Pre  num contains integral value to be converted
           bitStr is pointer to variable for bit string
      Post bit string stored in str
*/
void bin16 (uint16_t num, char* bitStr)
{
// Statements
   for (int i = 0; i < 16; i++)
        bitStr[i] = (char) ((num >> 15 - i) &
                     0X0001) + 48;

   return;
 } // bin16
```

```
Results:
Original value:  0000000001000000 (0x0040)
Shifted 1 right: 0000000000100000 (0x0020)
Shifted 2 right: 0000000000010000 (0x0010)
Shifted 4 right: 0000000000000100 (0x0004)
```

Dividing by 2

Let's start with something we know, decimal numbers. When we shift a decimal number one position to the right and insert a 0 on the left, we are in effect dividing by 10. If we shift it two places, we are dividing by 100. If we shift it three places, we are dividing by 1000. But actually, we are dividing by a power of 10—in our examples, 10^1, 10^2, and 10^3.

Applying the same principle to binary numbers, the right-shift operator divides by a power of 2. If we shift a binary number two places to the right, we are dividing by 4 (2^2). If we shift it three places, we are dividing by 8 (2^3). Table 14-6 shows the division pattern used with bit shifting.

$2^{\text{shift value}}$	Divides by	Shift Operator
1	2	>> 1
2	4	>> 2
3	8	>> 3
4	16	>> 4
…	…	…
n	2^n	>> *n*

TABLE 14-6 Divide by Shift

Bitwise Shift-Left Operator

The **bitwise shift left** (<<—precedence 11) is a binary operator that requires two integral operands (character or integer). The first operand is the value to be shifted. The second operand specifies the number of bits to be shifted.

Shifting binary numbers is just like shifting decimal numbers. If we have an eight-digit decimal number, and we shift it three places to the left, then the leftmost three digits are lost and three zero digits are added on the right. The binary shift operation is shown in Figure 14-3.

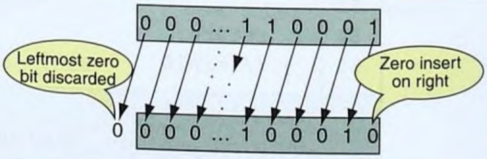


FIGURE 14-3 Shift-left Operation

Program 14-7 demonstrates the basic operation of the shift-left operator. Again, it uses the `bin16` function, which we have now included as a library.

PROGRAM 14-7 Simple Shift-left Demonstration

```
/* Demonstrate the bitwise shift-left operator.
     Written by:
     Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#include "bin16.c"

int main (void)
{
// Local Definitions
   uint16_t num  = 0x0031;
   uint16_t res;
   char     bitStr[17] = {"0"};

// Statements
   bin16 (num, bitStr);
   printf("Original value: %s (%#06x)\n", bitStr, num);

   res = num << 1;
   bin16 (res, bitStr);
   printf("Shifted 1 left: %s (%#06x)\n", bitStr, res);

   res =  num << 2;
   bin16 (res, bitStr);
   printf("Shifted 2 left: %s (%#06x)\n", bitStr, res);

   res = num << 4;
   bin16 (res, bitStr);
   printf("Shifted 4 left: %s (%#06x)\n", bitStr, res);

   return 0;
} // main
```

```
Results:
Original value: 0000000000110001 (0x0031)
Shifted 1 left: 0000000001100010 (0x0062)
Shifted 2 left: 0000000011000100 (0x00c4)
Shifted 4 left: 0000001100010000 (0x0310)
```

Multiplying by 2

Shift left is the opposite of shift right; multiply is the opposite of divide. It is reasonable to assume, therefore, that shifting left has the effect of multiplying by a power of 2. If you examine the results of Program 14-7 closely, you will note that is exactly what happens. When we shift a binary number two places to the left, we are multiplying by 4 (2^2). If we shift it three places, we are multiplying by 8 (2^3). Table 14-7 shows the multiplication pattern used with bit shifting.

$2^{\text{shift value}}$	Multiplies by	Shift Operator
1	2	<< 1
2	4	<< 2
3	8	<< 3
4	16	<< 4
…	…	…
n	2^n	<< *n*

TABLE 14-7 Multiply by Shift

Rotation

In computer science, we often need to rotate bits. For example, in security the creation of a hash digest of a message rotates and then scrambles bits to create a hash digest of data.

Rotation requires that bits be taken off one end of a value and moved to the other end. When we rotate bits left, we shift them off the left end of the value and insert them on the right. When we rotate right, we shift off the right end and insert them on the left. Figure 14-4 demonstrates the rotation of a 16-bit integer 4 bits to the right and then 4 bits to the left.

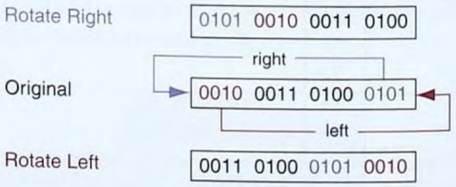


FIGURE 14-4 Right and Left Rotation

Note that the original number in hexadecimal is 0x2345. The rotated right number is 0x5234; the rotate left numbers is 0x3452.

Although there are no explicit operands for rotating left or right, the rotations can be coded using two shifts and an *or*. For example, to rotate the original 16-bit number in Figure 14-4, we use the following steps.

```
Original:                       0x2345
Shift right 4 bits:             0x0234
Shift left (16 - 4) bits:       0x5000
or'd results:                   0x5234
```

The rotate left operation is similar. Program 14-8 is a test driver for the rotate left and right functions.

PROGRAM 14-8 Rotate Left and Right Test Driver

```
/* Test driver for rotate left and right.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdint.h>

// Function Declarations
uint16_t rotate16Left  (uint16_t num , int n);
uint16_t rotate16Right (uint16_t num , int n);

int main (void)
{
// Local Declaration
   uint16_t num   = 0X2345;

// Statements
   printf("Original:      %#06X\n", num);
   printf("Rotated Left:  %#06X\n",
          rotate16Left (num, 4));
   printf("Rotated Right: %#06X\n",
          rotate16Right(num, 4));
} // main

/* ================== rotate16Left ===================
   Rotate 16 bit fixed size integer left n bits.
      Pre  num is a fixed size 16-bit integer
      Post num rotated n bits left
*/
uint16_t rotate16Left (uint16_t num , int n)
{
   return  ( ( num  << n ) | ( num >> 16 - n ) );
```

continued

PROGRAM 14-8 Rotate Left and Right Test Driver *(continued)*

```
} // rotate16Left

/* =================== rotate16Right ==================
   Rotate 16 bit fixed size integer right n bits.
      Pre  num is a fixed size 16-bit integer
      Post num rotated n bits right
*/
uint16_t rotate16Right (uint16_t num , int n)
{
   return  ( ( num >> n ) | ( num << 16 - n ) );
} // rotate16Right
```

```
Results:
Original:      0X2345
Rotated Left:  0X3452
Rotated Right: 0X5234
```

14.4 Masks

In many programs, bits are used as binary flags: 0 is off, and 1 is on. To set and test the flags, we use a bit mask. A **mask** is a variable or constant, usually stored in a byte or short integer, that contains a bit configuration used to control the setting or testing of bits in a bitwise operation. The bits are numbered from the least significant bit (rightmost), starting at 0, as shown in Figure 14-5.

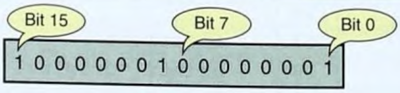


FIGURE 14-5 Bit Mask in a 16-bit Integer

Creating Masks

Masks can be created statically or dynamically. Static masks are generally literals. For example, to create the mask in Figure 14-5, we could code the constant `0x8101`. Dynamic masks are created using the bit manipulation operators: shift, *and*, *or*. In this section, we examine the basic techniques for creating dynamic masks, that is, masks that may change as the program executes.

One-Bit Masks

Given a byte flag in which bit 5 is an error flag, we need to create a mask to test it. To test bit 5, the mask must have bit 5 set to 1 and the rest of the bits to 0. This is easily done, as shown below.

```
uint8_t mask;
mask = 0x01 << 5;

00000001 << 5 → 00100000
```

We start with a constant binary 1. This places a one bit in the rightmost position. We then shift the one bit to the left five positions. The result is shown in the previous example.

An easier way to control the location of the bit flags is to name them. This is easily done with a defined constant, as shown in the next example.

```
#define FLAG0    0
#define FLAG1    1
...
#define FLAG5    5

mask = 0x01 << FLAG5;
```

By using named flag positions, we free ourselves from remembering their locations and at the same time make the code more readable. Of course, in an application, we would use meaningful names, such as `OVERFLOW_FLAG`, not `FLAG1`.

Two-Bit Masks

Creating a mask with two flags requires that we set 2 bits on. The easiest way to do this is to set two individual flags and then *or* them.

```
uint8_t mask;
mask    = (0x01 << FLAG1) | (0x01 << FLAG5);
```

The result of the above code would set the mask as shown below.

```
00100010
```

Setting Bit Range Masks

To set a range of bits requires a loop. The following example sets bits 7 through 3.

```
uint8_t mask  = 0x00;

for (i = 3; i <= 6; i++)                              // 01111000
     mask = mask | 0x01 << i;
```

Complement Masks

To create a mask that turns off a specific bit—that is, that complements it—we must set the bit to 0 while setting the rest of the bits in the mask to 1. This requires two statements. In the first statement, we set only the desired bit on. In the second step, we complement the mask, changing all zero bits to 1 and all one bits to 0. The code is shown below.

```
mask = 0x01 << FLAG1;
mask = ~mask;
```

Using Masks

Now that we've seen how to create masks with various bit flags set, let's look at some examples that use them.

Turning Bits Off

One of the applications of the bitwise *and* operator is turning bits off—that is, forcing selected bits to zero. For example, an Internet address uses the format x.y.z.t as we have seen before. A mask in the form /n, often called prefix, is used to find the network or subnet address to which a computer is attached. The prefix /n defines the number of leftmost contiguous 1s out of 32 bits. A prefix of /20 means that the leftmost 20 bits are 1s and the rightmost bits are 0s. This is demonstrated in the next example.

```
Computer address: 01111011010011100001100100001101
Mask:             11111111111111111111000000000000
Network address:  01111011010011100001000000000000
```

Program 14-9 determines a network address given a computer host address. To be flexible, it asks the user to enter the size of the prefix.

PROGRAM 14-9 Determine Network Address

```
/* Given a host address and the size of the prefix,
   determine its network address.
```

continued

PROGRAM 14-9 Determine Network Address *(continued)*

```
       Written by:
       Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

int main (void)
{
// Local Declarations
   unsigned int comAddr[4];
   unsigned int mask[4];
   unsigned int netAddr[4];
   uint32_t     comAd  = 0;
   uint32_t     mask32 = 0;
   uint32_t     netAd  = 0;
   int          prefix;

// Statements
   printf ("Enter host address <x.y.z.t>: ");
   scanf ("%d%*c%d%*c%d%*c%d",
         &comAddr[3], &comAddr[2], &comAddr[1],
&comAddr[0]);

   printf ("Enter prefix: ");
   scanf ("%d", &prefix);

   // Convert address to a 32-bit computer address
   for (int i = 3; i >= 0; i--)
        comAd = comAd * 256 + comAddr[i];

   // Create a 32-bit mask
   for (int i = 32 - prefix; i < 32; i++)
        mask32 = mask32 | (1 << i);

   // AND to get a 32-bit Network Address
   netAd = comAd & mask32;

   // Change mask into the form x.y.z.t
   for (int i = 0; i < 4; i++)
       {
        mask[i] = mask32 % 256;
        mask32  = mask32 / 256;
```

continued

PROGRAM 14-9 Determine Network Address *(continued)*

```
45            } // for
46
47        // Change IP address into the form x.y.z.t
48        for (int i = 0; i < 4; i++)
49            {
50             netAddr[i] = netAd % 256;
51             netAd      = netAd / 256;
52            } // for
53
54        // Print Addresses
55        printf ("\nAddresses:\n");
56        printf ("Computer Address: ");
57        printf ("%d.%d.%d.%d\n",
58                comAddr[3], comAddr[2],
   comAddr[1],comAddr[0]);
59
60        printf ("Mask:              ");
61        printf ("%d.%d.%d.%d\n",
62                mask[3], mask[2], mask[1], mask[0]);
63
64        printf ("Net Address:       ");
65        printf ("%d.%d.%d.%d\n",
66                netAddr[3], netAddr[2], netAddr[1],
   netAddr[0]);
67        return 0;
68    }  // main
```

```
Results:
Enter host address <x.y.z.t>: 123.45.78.12
Enter prefix: 18

Addresses:
Computer Address: 123.45.78.12
Mask:             255.255.192.0
Net Address:      123.45.64.0
```

Program 14-9 Analysis

Several parts of this program need to be studied carefully. First, note that the variable for the network address is an array of four integers. In statement 23, we read the address in dotted decimal format into the array.

The *for* loop in statement 30 converts the components of the address into a 32-bit address. Each component of the address is considered one byte in the final address. Since each byte has a value of 256 (2^8), we compute the address as follows:

$$byte[0] + byte[1] \times 256^1 + byte[2] \times 256^2 + byte[3] \times 256^3$$

Once we have the dotted-decimal address converted to a 32-bit address, we must create a prefix mask by setting a 32-bit mask to /n leading 1 bits. This is done in the *for* loop in statement 34.

Finally, the network address and mask are converted back to the dotted-decimal format. Each byte is determined by taking modulo 256 of part of the network address. To get the next byte in the address, we then divide by 256. This logic is seen in the *for* loops in statement 41 and 48.

Turning Bits On

One common application of bit inclusive *or* is **turning bits on**—that is, forcing selected bits in a field to one. To force to one, all that we need to do is to construct a mask with the desired one bit set on. This guarantees that the result will have a one bit in that location.

A mask can be used to find the last address in a block of network addresses. The last address is used for limited broadcasts. If a computer wants to send a message to all computers inside its network, it uses this address.

To find the last address, we *or* the complement of a prefixed mask with any address in the block. Conversely, to find the first address in the network block, we *and* the mask with any address. Program 14-10 determines the last address in a network.

PROGRAM 14-10 Determine Last Address in a Network

```
/* Determine the last address in a broadcast network.
      Written by:
      Date:
*/
#include <stdio.h>
#include <math.h>
#include <stdint.h>

int main (void)
{
// Local Declarations
   unsigned int  comAddr[4];
   unsigned int  mask[4];
   unsigned int  broadAddr[4];

   uint32_t comAd    = 0;
   uint32_t mask32   = 0;
   uint32_t broadAd  = 0;
   int      prefix;

// Statements
   printf ("Enter host address <x.y.z.t>: ");
   scanf ("%d%*c%d%*c%d%*c%d",
```

continued

PROGRAM 14-10 Determine Last Address in a Network *(continued)*

```
             &comAddr[3], &comAddr[2],
             &comAddr[1], &comAddr[0]);
   printf("Enter prefix: ");
   scanf ("%d", &prefix);

   // Convert address to 32-bit computer address
   for (int i = 3; i >= 0; i--)
       comAd = comAd * 256 + comAddr[i];

   // Create 32-bit prefix mask
   for (int i = 32-prefix; i < 32; i++)
       mask32 = mask32 | (0x0001 << i);

   // And to get a 32-bit Network Address
   broadAd = comAd | (~mask32);

   // Change the mask into the form x.y.z.t
   for (int i= 0; i < 4; i++)
      {
       mask[i]= mask32 % 256;
       mask32 = mask32 / 256;
      } // for

   // Change the IP address to the form x.y.z.t
   for (int i= 0; i < 4; i++)
      {
       broadAddr[i] = broadAd % 256;
       broadAd      = broadAd / 256;
      } // for

   printf ("\nPrinting Addresses\n");
   printf ("Computer Address:  ");
   printf ("%d.%d.%d.%d\n",
           comAddr[3], comAddr[2],
           comAddr[1], comAddr[0]);
   printf ("Mask:              ");
   printf ("%d.%d.%d.%d\n",
           mask[3], mask[2], mask[1], mask[0]);

   printf ("Broadcast Address: ");
   printf ("%d.%d.%d.%d\n",
           broadAddr[3], broadAddr[2],
           broadAddr[1], broadAddr[0]);
   return 0;
```

PROGRAM 14-10 Determine Last Address in a Network *(continued)*

```
}  // main
```

```
Results:
Enter host address <x.y.z.t>: 123.45.78.12
Enter prefix: 18

Printing Addresses
Computer Address:   123.45.78.12
Mask:               255.255.192.0
Last Address:       123.45.127.255
```

Program 14-10 Analysis Let's compare Program 14-9 and Program 14-10. In Program 14-9, we calculate the first address in a block. In Program 14-10, we calculate the last address in a block. If we subtract these addresses (in base 256) and add 1, we get the total addresses in the block. Looking at the program results, we see that the first address is 123.45.64.0 (Program 14-9) and the last address is 123.45.127.255 (Program 14-10). The difference is

$$(127-64) * 256 + (255 - 0) + 1 = 16,384$$

We have 16,384 addresses in this block. We could have found this result using the prefix (18).

$$2^{32\text{-prefix}} = 2^{32-18} = 2^{14} = 16,384$$

Flipping Bits

One of the applications of bitwise **exclusive *or*** is **flipping bits**—that is, forcing 0s to 1s and 1s to 0s, called force to change. The resulting bit is 1 when 1 bit is a 0 and the other is a 1; if both bits are 0 or both are 1, the result is 0. To force a bit to change: therefore, the forcing bit in the mask is set to 1; bits that are to be unchanged are set to 0.

For example, assume that we want to change the 5 leftmost bits of a number stored as an 8-bit integer. We create the mask with 5 one bits and 3 zero bits, as shown below. (Y indicates a changed bit.)

```
operator ^    number     XXXXXXXX
              mask       11111000
                         --------
              result     YYYYYXXX
```

Dividing Polynomials

As a more advanced example of flipping bits, let's examine how we can use fixed-size integers to represent polynomials. Polynomials with coefficients of

1 or 0 are used in computer science for error detection, encoding, and other applications.

A fixed-size integer can represent a polynomial if we think of 0 and 1 as coefficients of the polynomial. For example, the bit stream `01001001` can represent a polynomial of degree 6 or $x^6 + x^3 + 1$. In this polynomial, the coefficients of x^7, x^5, x^4, x^2, and x^1 are 0 and the coefficients x^6, x^3, and x^0 are 1. This relationship is shown in Figure 14-6.

In general a bit stream of length n can represent a polynomial of length $n - 1$. So a data type of *uint16_t* can represent a polynomial of up to degree 15.

A common polynomial operation is division. The division of one polynomial by another is shown in Figure 14-7.

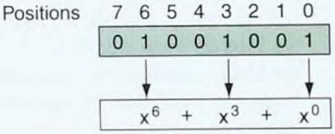
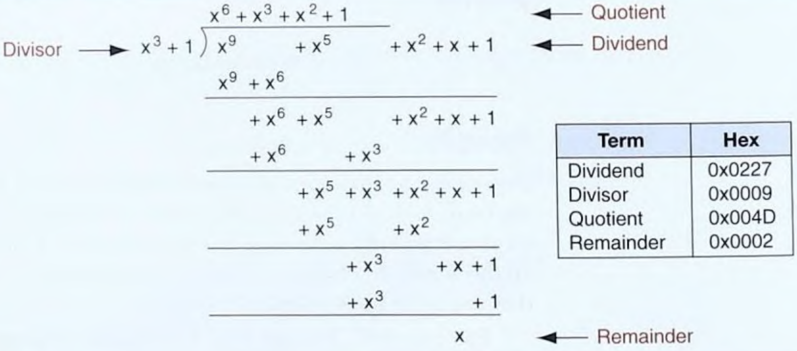


FIGURE 14-6 Polynomial Coefficients



Term	Hex
Dividend	0x0227
Divisor	0x0009
Quotient	0x004D
Remainder	0x0002

FIGURE 14-7 Polynomial Division

Although the division of polynomials is similar to the division of numbers we learned in elementary school, there are two differences:

1. Because we use only coefficients 0 and 1, if we have two terms of the same exponent, the result is zero $(1 + 1 = 0)$. In other words, polynomial division uses modulo 2 arithmetic on the coefficients.
2. Subtraction and addition of the coefficients are actually the same (exclusive *or*) because we are interested in only coefficients of 0 and 1.

Note that if we multiply the quotients by the divisor and add it with the remainder, we get the dividend. However, if we get two identical coefficients (such as x^6), we cancel them because 1 + 1 = 0.

Program 14-11 demonstrates polynomial division.

PROGRAM 14-11 Polynomial Division

```
/* Demonstrate polynomial division.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdint.h>

// Function Declaration
int degree (uint16_t);

int main (void)
{
// Local Declaration
   uint16_t dvdn = 0X0227;
   uint16_t dvsr = 0X0009;
   uint16_t qtnt = 0X0000;
   uint16_t rmdr;
   uint16_t q;
   int      dgre;

// Statements
   printf ("Dividend:  %#06X\n", dvdn);
   printf ("Divisor:   %#06X\n", dvsr);

   rmdr = dvdn;
   while ((dgre = degree (dvdn) - degree (dvsr)) >= 0)
      {
       q    = 1 << dgre;
       rmdr = dvdn ^ (dvsr << degree (q));
       qtnt = qtnt | q;
       dvdn = rmdr;
      } // while

   printf ("Quotient : %#06X\n", qtnt);
   printf ("Remainder: %#06X\n", rmdr);
   return 0;
}  // main

/* Determine degree of polynomial represented by
```

continued

PROGRAM 14-11 Polynomial Division *(continued)*

```
   a fixed-length 16-bit variable.
       Pre  poly represents a polynomial
       Post degree returned
*/
int degree (uint16_t  poly)
{
// Local Declarations
   uint16_t mask = 0X0001;
   uint16_t temp;
   int      pos= -1;

// Statements
    for (int i = 0; i <16; i++)
       {
        temp = poly >> i;
        if ((temp & mask) == 1)
             pos = i;
       } // for
   return pos;
}  // degree
```

```
Results:
Dividend:  0X0227
Divisor:   0X0009
Quotient : 0X004D
Remainder: 0X0002
```

Program 14-11 Analysis We use a function to determine the degree of the polynomial. In *main* we use a *while* loop to simulate the division. The variable `poly` is the term in the quotient that is obtained in each step. Note that the multiplication of a term such as x^n by the division is done by shifting the divisor *n* bits.

14.5 Software Engineering

In Chapter 12, we looked at what makes a good function. In this section, we look at how you design good programs.

Payroll Case Study

To provide a discussion focus, we will use a payroll program. Although our example is rather simple, it does contain all the elements involved in designing a program. The description of the payroll program is shown in Figure 14-8.

Payroll Case Study

1. Requirements:

   Given employees and their hours worked, compute net pay and record all payroll data for subsequent processing, such as W2 statements. Prepare paychecks and a payroll ledger.

   Maintain data on a sequential payroll file.

2. Provide for the following nonstatutory deductions:

   a. Health plan
   b. United Way
   c. Union dues

3. The payroll data are:

   a. Employee number
   b. Pay rate
   c. Union member flag
   d. United Way contribution
   e. Exemptions

4. Maintain the following year-to-date totals:

   a. Earnings
   b. FICA taxes
   c. SDI taxes
   d. Federal withholding
   e. State withholding
   f. Health plan fees
   g. United Way donations
   h. Union dues

5. Algorithms:

   a. Gross Pay = (Reg Hrs * Rate) + (OT Hours * Rate * 1.5)
   b. FICA Taxes = (Gross Pay*FICA Rate) if less than Max FICA
   c. SDI Taxes = (Gross Pay * SDI Rate) if less than Max SDI
   d. Taxable Earnings = (Gross Pay – (Exemptions * Exemption Rate))

continued

FIGURE 14-8 Requirements for Payroll Case Study

Payroll Case Study

e. Federal Taxes = (Taxable Earnings * Federal TaxRate)
f. State Taxes = (Taxable Earnings * State Tax Rate)
g. Net Pay = (Gross Pay – (FICA Taxes + SDI Taxes +
    Federal Taxes + State Taxes + Health Fee +
    United Way Donation + Union Dues))

FIGURE 14-8 Requirements for Payroll Case Study *(continued)*

Program Design Steps

As you know by now, a program is developed in seven steps:

1. Determine requirements.
2. Determine data structures.
3. Build structure charts.
4. Create test cases.
5. Write and unit test the programs.
6. Test system.
7. Implement system in production.

Our interest here is only in the third step: Build the structure charts. A few general comments are in order, however. The second and third steps are often reversed or done concurrently. Which one is done first is not of major consequence, as long as they are done before the fourth step.

Many programmers think that test cases should be built after the program is written. Good programmers know better. By creating test cases based on the requirements (Step 1) and your design (Steps 2 and 3), you will understand the problem better. You will even find occasions when you change your design based on what you learned creating test cases.

This does not mean that you will be done with creating test cases at Step 4; you just start there. You will develop more test cases *while you are writing the program,* and you will create still more as you conduct unit testing.

Structure Chart Design

A good program starts with a good design, as reflected in the structure chart. By now, you should have progressed to the point at which you are designing your programs before you start coding; that is, you are creating your structure chart first.

One tool to design the structure chart is known as **transform analysis**. Transform analysis is a design technique that identifies the processes in a program as input, process, and output and then organizes them around one or more processes that convert inputs to outputs. These conversion processes are known as the **central transforms**. Having determined the first-cut design, you repeat the process, decomposing the identified modules into subtasks using transform analysis.

This is a good design technique whenever a program reads, processes, and writes data, which covers the majority of programs. Although it is usually used in conjunction with another tool known as a data flow diagram, you can use it independently.

> Good programs start with a good structure chart design.

Recognize, however, that transform analysis is an approach to the design of a program. It is not a cookbook that leads to the same results every time. Different programmers using the same steps will arrive at different designs.

Programmers use six steps in designing a structure chart:

1. Determine program modules.
2. Classify modules.
3. Construct preliminary structure chart.
4. Decompose modules.
5. Complete structure chart.
6. Validate design.

Determine Modules

The first step in program design is to identify the processes that the program will use. This is usually done by reviewing the program specification and identifying the tasks it needs to accomplish. A review of our payroll case identifies the following tasks. (The references are to the case study description in Figure 14-8.)

1. Read hours worked (1).
2. Compute pay (1).
3. Maintain payroll master file (1).
4. Prepare paychecks (1).
5. Prepare payroll ledger (1).
6. Calculate nonstatutory deductions (2).
7. Calculate year-to-date totals (4).
8. Calculate gross pay (5).
9. Calculate taxes (5).

Remember the rule that each module should do only one thing. You might be tempted to start with a module to prepare output instead of prepare paychecks and prepare payroll ledger. And in fact, you may well end up with an intermediate module that combines the preparation of all reports. But at this point, we want to keep different things separate as much as possible. On the other hand, experience indicates that to have separate modules for calculating FICA taxes, SDI taxes, federal taxes, and state taxes is unnecessary.

Classify Modules

In transform analysis, we look for the central transforms—that is, the module(s) that turns inputs into outputs. To identify these transforms, we classify each identified task, which now represents a module in the structure chart, as afferent, efferent, or transform. A module is an **afferent module** if its processing is directed toward the central transform. Another way of looking at this concept is to say that the module is a gatherer of data. **Efferent modules** direct data away from the central processing or toward the outputs of the program. (Remember that input comes before output and *a* comes before *e*; therefore, afferent is input and efferent is output.) A **transform module** is balanced; that is, it has data flowing both in and out. The concepts of afferent, efferent, and transform are shown in Figure 14-9.

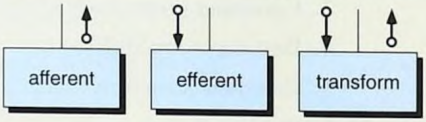


FIGURE 14-9 Afferent, Efferent, and Transform Modules

Table 14-8 classifies each of the modules identified above as afferent, efferent, or transform.

Module	Afferent	Efferent	Transform
Read hours worked	✓		
Compute pay			✓
Maintain payroll master file	✓	✓	
Prepare paychecks		✓	
Prepare payroll ledger		✓	
Calculate nonstatutory deductions			✓
Calculate year-to-date totals			✓
Calculate gross pay			✓
Calculate taxes			✓

TABLE 14-8 Classification of Payroll Modules

We have two comments about this classification. Note that "maintain payroll master file" is classified as both input and output. This is because it is a master file. We will need to read it to get the employee personnel and history data, and we will need to write the updated data after they have been calculated. Therefore, it is really two modules: one to read the master and one to write the master.

Second, there are many transform modules. This will result in a large fan out, which is not desirable. **Fan out** is the number of submodules emanating from a module. We will need to reduce the fan out later.

Construct Structure Chart

At this point, you are ready to construct the first-cut structure chart. Transform analysis structure charts are organized with inputs on the left, transforms in the center, and outputs on the right. This organization is shown in Figure 14-10.

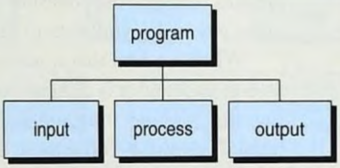


FIGURE 14-10 Basic Structure Chart Organization

For the first-cut structure chart, place all afferent modules below the input block, all transform modules below the process block, and all efferent modules below the output block. In this process, analyze each module to determine if it needs to be called before or after the other processes at the same level. Those that need to be called first are placed on the left, and those that need to be called last are placed on the right. The resulting structure chart is shown in Figure 14-11.

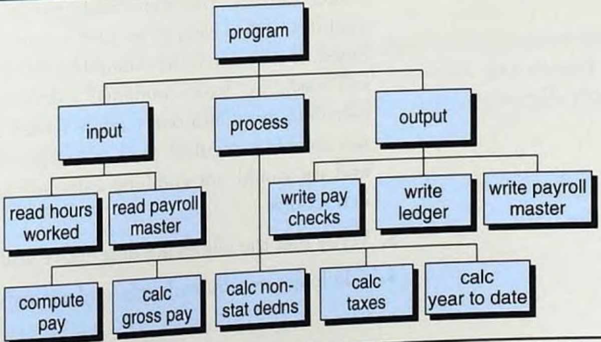


FIGURE 14-11 First-cut Structure Chart

Up to this point, we have been almost algorithmic; that is, we have exercised little judgment. We now need to analyze the preliminary structure chart to see if it makes sense. This is done by asking a simple question, "What do we mean by...?" For example, "What do we mean by compute pay?" The answer is that we must take hours worked and multiply them by the pay rate, also considering overtime and so forth. Or, to put it another way, we compute gross pay. But this is already a module in the structure chart, so these two modules are the same. We will therefore delete compute pay since it is a less specific description than calculate gross pay.

Decompose Modules

To decompose modules, we look at the cohesion of each module. For example, when we ask what we mean by calculate taxes, the answer is calculate federal taxes and calculate state taxes. Since we are dealing with two different entities (things), the cohesion of this module is communicational; it uses the gross pay and payroll data to calculate the different taxes.

When we find that a module is doing more than one thing, or that it is so complex that it is difficult to understand, then we need to consider breaking the module into submodules. This refinement of the modules was named **stepwise refinement** by Niklaus Wirth, the creator of Pascal.[1] It refines the processes in a module until each module is at its most basic, primitive meaning.

Decomposition continues until the lowest levels in our structure chart are all functionally cohesive and easily understood.

Complete Structure Chart

At this point we have the nucleus of the structure chart complete, and all we have to do is add the finishing touches. These steps are almost mechanical.

1. Identify any common processes with a cross-hatch in the lower right corner. In our payroll case, there are none.
2. Consider adding intermediate (middle-level) modules if necessary to reduce fan out. This step should not be done arbitrarily, however. If the modules next to each other have a common entity, then they can be combined. If they don't, they should be left separate. For example, in the payroll case, we have combined calculate nonstatutory deductions and calculate taxes into one module called calculate deductions. We would not combine calculate gross pay with calculate nonstatutory deductions, and we would not combine calculate taxes with calculate year-to-date deductions.
3. Verify that the names are descriptive and meaningful for their processes.
4. Add loops, conditional calls, and exclusive *or* designators.

[1] Niklaus Wirth, "Program Development by Stepwise Refinement," *Communications of the ACM*, 14, no. 4 (April 1971).

5. Add input/output modules, if not already present.
6. Add initialization and end of job modules.
7. Add error routines (if necessary).
8. Add data flows and flags as required.

The completed design for the payroll case study is shown in Figure 14-12.

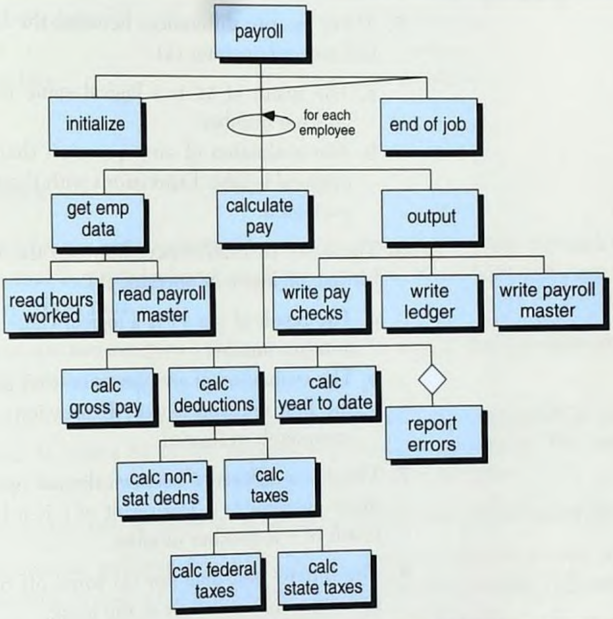


FIGURE 14-12 Final Payroll Structure Chart

Validate the Design

At this point, you are ready to validate the design with a structured walk-through. Before you convene the review board, however, you should review your design once more by repeating all the design steps, especially the functional decomposition step.

14.6 Tips and Common Programming Errors

1. We recommend that bitwise operators be used only with unsigned fixed-size operands.
2. To use fixed-size integers, we need to include the *stdint.h* file header file.
3. The bitwise *and* operator is only one ampersand (`&`), not two.
4. The bitwise inclusive *or* operator is only one bar (|), not two.
5. There are two differences between the logical *and* operator (`&&`) and the bitwise *and* operator (`&`).
   a. The result of `&&` is a logical value (*true* or *false*). The result of `&` is another number.
   b. The evaluation of an expression that uses `&&` terminates if the first operand is *false*. Expressions with the `&` operator are always completely evaluated.
6. There are two differences between the logical *or* operator (||) and the bitwise inclusive *or* operator (|).
   a. The result of the || is a logical value (*true* or *false*). The result of | is another number.
   b. The evaluation of an expression that uses the || operator terminates if the first operand is *true*. Expressions with the | operator are always completely evaluated.
7. There is a difference between the *not* operator (`!`) and the one's complement operator (~). The result of `!` is a logical value (*true* or *false*). The result of ~ is another number.
8. The bitwise *and* operator (`&`) turns off only the corresponding bits in a data item that match 0s in the mask.
9. The bitwise *or* operator (|) turns on only the corresponding bits in a data item that match 1s in the mask.
10. The bitwise exclusive *or* operator (^) flips only the corresponding bits in a data item that match 1s in the mask.
11. The bitwise complement operator (~) flips all the bits in a variable.
12. Avoid using the shift operators with the signed number.
13. The shift-left operator (<<) multiplies a number by a power of 2 (2^n), not simply by 2.
14. The shift-right operator (>>) divides a number by a power of 2 (2^n), not simply by 2.
15. Rotating bits requires a combination of shift right and shift left operations.
16. The difference between bitwise exclusive *or* (^) and the one's complement (~) is that the ~ operator flips all the bits; the ^ operator flips only the specific bits in the mask.
17. To add *n*-bit numbers in one's complement, we use modulo $2^n - 1$.

14.7 Key Terms

afferent module
bitwise *and* operation
bitwise *exclusive or* operation
bitwise *inclusive or* operation
bitwise operation
central transform
efferent module
fan out
flipping bits
turning bits on
turn bits off
mask
one's complement
bitwise shift left operation
bitwise shift right operation
stepwise refinement
transform analysis
transform modules

14.8 Summary

- C is a proper language for system programming because it contains operators that can manipulate data at the bit level. These operators are called bitwise operators.

- There are two categories of bitwise operators: logical bitwise operators and shift bitwise operators.

- The bitwise *and* (`&`) is a binary operator applied to two operands of integral value. It does a bit-by-bit comparison of the two operands. The result of each comparison is 1 if both bits are 1s. It is 0 otherwise.

- The bitwise *and* operator (`&`) is used to turn off masked bits in an operand.

- The bitwise inclusive *or* (`|`) is a binary operator applied to two integral operands. It does a bit-by-bit comparison of its two operands. The result of each comparison is 0 if both bits are 0s. It is 1 otherwise.

- The bitwise inclusive *or* operator (`|`) is used to turn on mask bits in an operand.

- The bitwise exclusive *or* (`^`) is a binary operator applied to two integral operands. It does a bit-by-bit comparison of its two operands. The result of each comparison is 1 only if the two bits are different. If they are the same, the result is 0.

- The bitwise exclusive *or* operator (`^`) is used to flip masked bits in an operand.

- The bitwise complement operator (`~`) is a unary operator. It changes the value of each bit from 0 to 1 or from 1 to 0.

- The second operand in the bitwise *and*, bitwise inclusive *or*, and bitwise exclusive *or* is often called a mask.

- Bitwise shift operators are binary operators used to shift data to the right or the left. The second operand in these operators is an integer that defines the number of bits to be shifted.

- ❑ The bitwise left-shift operator (<<) shifts the bits in the first operand to the left as many bits as is defined by the second operand. This effectively multiplies a number by 2^n, where n is the number of bits to be shifted.
- ❑ The bitwise right-shift operator (>>) shifts the bits in the first operand to the right as many bits as is defined by the second operand. This effectively divides a number by 2^n, where n is the number of bits to be shifted.
- ❑ The C language has no operator to rotate bits in a data item. Rotation can be accomplished by *or*'ing a shift-left operation and a shift-right operation.

14.9 Practice Sets

Review Questions

1. The result of bitwise *and*'ing of two bits is 1 if ________________.

   a. both bits are 0s.
   b. both bits are 1s.
   c. bits are different

2. The result of bitwise inclusive *or*'ing of two bits is 0 if ________________.

   a. both bits are 0s.
   b. both bits are 1s.
   c. bits are different

3. The result of bitwise exclusive *or*'ing of two bits is 1 if ________________.

   a. both bits are 0s.
   b. both bits are 1s.
   c. bits are different

4. To change every bit in a variable from a 1 to a 0 or from a 0 to a 1, we need to use the ________________ operator.

   a. &
   b. |
   c. ^
   d. ~

5. The ________________ operator can be used to turn masked bit on.

   a. &
   b. |
   c. ^
   d. ~

6. The ________________ operator can be used to turn masked bit off.

   a. &
   b. |

c. ^
d. ~

7. The ______________ operator can be used to flip masked bit.

a. &
b. |
c. ^
d. ~

8. The bitwise ______________ operator is used to divide a data item by a power of 2.

a. &
b. |
c. ^
d. >>
e. <<

9. In program design, a function that collects data that are directed toward a central transform is classified as a(n) ______________ module.

a. afferent
b. efferent
c. input
d. output
e. transform

10. To find the first address in a block of address, we need to use the ______________ operator with the network mask.

a. &
b. |
c. ^
d. >>
e. <<

11. To find the last address in a block of addresses, we need to use the ______________ operator with the complement of network mask.

a. &
b. |
c. ^
d. >>
e. <<

Exercises

12. Show the value of each of the following binary numbers in hexadecimal:

a. `11011001`
b. `11111111`
c. `1111000100011111`
d. `0000000011100001`

13. Show the value of each of the following hexadecimal numbers in binary:

    a. `0x37`
    b. `0xAB`
    c. `0x0237`
    d. `0xA234`

14. Determine the result of the following operations (operands are in binary):

    a. `~11011001`
    b. `~11111111`
    c. `~11110001`
    d. `~00000000`

15. Determine the result of the following operations (operands are in binary):

    a. `11111100 & 00111111`
    b. `11111111 & 10101010`
    c. `00000000 & 11111001`
    d. `10101010 & 11100001`

16. Determine the result of the following operations (operands are in binary):

    a. `11111100 | 00111111`
    b. `11111111 | 10101010`
    c. `00000000 | 11111001`
    d. `10101010 | 11100001`

17. Determine the result of the following operations (operands are in binary):

    a. `11111100 ^ 00111111`
    b. `11111111 ^ 10101010`
    c. `00000000 ^ 11111001`
    d. `10101010 ^ 11100001`

18. When numbers are represented in hexadecimal, we can simplify the result of one's complement (~) operation. The result of complementing each digit is 15 minus the value of that digit (5 = ~A). Using the above short-cut rule, find the result of the following operations:

    a. `~0x02A1`
    b. `~0xF305`
    c. `~0xE2A8`
    d. `~0x0FFA`

19. When numbers are represented in hexadecimal, we can use some rules that simplify the bitwise *and* (`&`) operation in three cases. If one of the digits is 0, the result is 0 (0 = 5 & 0). If one of the digits is F, the result is the other digit (3 = F & 3). If two digits are the same, the result is one of them. (A = A & A). Using the above short-cut rules, find the result of the following operations:

    a. `0xC201 & 0x02AF`
    b. `0xA240 & 0xF005`

c. `0xE205 & 0xE2A0`

d. `0xA531 & 0x0FF0`

20. When numbers are represented in hexadecimal, we can use some rules that simplify the bitwise inclusive *or* (|) operation in three cases. If one of the digits is 0, the result is the other digit (5 = 5 | 0). If one of the digits is F, the result is F (F = F | 3). If two digits are the same, the result is one of them. (A = A | A). Using the above short-cut rules, find the result of the following operations:

a. `0xC201 | 0x02AF`

b. `0xA240 | 0xF005`

c. `0xE205 | 0xE2A0`

d. `0xA531 | 0x0FF0`

21. When numbers are represented in hexadecimal, we can use some rules that simplify the bitwise exclusive *or* (^) operation in three cases. If one of the digits is 0, the result is the other digit (5 = 5 ^ 0). If one of the digits is F, the result is 15 minus the other digit (9 = F ^ 6). If two digits are the same, the result is 0 (0 = A ^ A). Using the above short-cut rules, find the result of the following operations:

a. `0xC201 ^ 0x02AF`

b. `0xA240 ^ 0xF005`

c. `0xE205 ^ 0xE2A0`

d. `0xA531 ^ 0x0FF0`

22. Find the value of the following expressions:

a. `0x13 << 2`

b. `0x13 >> 2`

c. `0x13A2 << 4`

d. `0xB23E >> 2`

23. Show the mask and the operator that turn on the most significant bit of an 8-bit integer. (Remember, the bits are numbered from the right, starting with 0.) Give the answer in binary and hexadecimal form.

24. Show the mask and the operator that turn off the most significant bit of an 8-bit integer. (Remember, the bits are numbered from the right, starting with 0.)

25. Find the mask that complements the values of the first and third bits of an 8-bit integer. (Remember, the bits are numbered from the right, starting with zero.) What operator should be used?

26. Find the mask that when used with the `&` operator sets the second and fourth bits of an 8-bit integer to zero. (Remember, the bits are numbered from the right, starting with 0.)

27. Find the mask that when used with the | operator sets the second and fourth bits of an 8-bit integer to 1. (Remember, the bits are numbered from the right, starting with 0.)

28. What is the result of the following expressions if *a* is an 8-bit unsigned integer?

    a. `a & ~a`
    b. `a | ~a`
    c. `a ^ ~a`
    d. `a & 0xFF`
    e. `a & 0x00`
    f. `a | 0xFF`
    g. `a | 0x00`
    h. `a ^ 0xFF`
    i. `a ^ 0x00`

29. Show the binary and hexadecimal representation of following polynomial using 16-bit unsigned integer: $x^{11} + x^6 + x^3 + x + 1$

30. Given two polynomials $x^4 + x^3 + 1$ and $x^6 + x^3 + x + 1$

    a. Add them using modulo 2 arithmetic for coefficients.
    b. Use 8-bit unsigned integers (in hexadecimal) to represent each polynomial and apply addition using *exclusive or* operations.
    c. Compare the result of part a and b.

31. Given two polynomials $x^5 + x^2 + 1$ and $x^6 + x^4 + x^2 + 1$

    a. Subtract the first polynomial from the second using modulo 2 arithmetic for coefficients.
    b. Use 8-bit unsigned integers (in hexadecimal) to represent each polynomial and apply subtraction using exclusive *or* operations.
    c. Compare the result of parts a and b.

32. Given two polynomials $x^{12} + x^4 + x^3 + 1$ and $x^6 + x^3 + x + 1$

    a. Divide the first polynomial by the second using modulo 2 arithmetic for coefficients. Find the quotient and the remainder.
    b. Use 16-bit unsigned integers (in hexadecimal) to represent each polynomial and use Program 14-11 to find the quotient and the remainder.
    c. Compare the result of part a and b.

Problems

33. Write a function that sets a specified bit in a 16-bit unsigned integer to 1. The function is to have two parameters: The first parameter is the integer to be manipulated and the second is the bit location, relative to the least significant bit, that is to be turned on. Your test program should start with 0 and turn on several bits in turn.

34. Rework Problem 33 to turn off (set to zero) the specified bit.

35. Rework Problem 33 to flip the specified bit.

36. Write a function that flips the bits in an 16-bit unsigned integer.

37. Write a function that accepts two polynomials (with 0 and 1 coefficient) in the form of two 16-bit unsigned integers and returns the result of adding the two in the form of a 16-bit unsigned integer. Use hexadecimal notations. Can the same function be used for subtracting the two polynomials? If yes, does the order of two polynomials make a difference?

38. Write a function that returns the first (leftmost) hexadecimal digit from a 32-bit unsigned integer.

39. Rework Problem 38 to return the last (rightmost) digit.

40. Write a function that complements the first (leftmost) hexadecimal digit in a 32-bit unsigned integer. Note that the complement of a hexadecimal digit is the 15 minus that digit.

41. Rework Problem 40 to complement the last (rightmost) digit.

42. Write a function that changes the first (leftmost) hexadecimal digit in a 32-bit unsigned integer. The function is to have two parameters: The first is the integer to be manipulated, the second the replacement digit.

43. Rework Problem 42 to change the last (rightmost) digit.

Projects

44. Modify Program 14-5 for checksum calculation at the sender site to a checksum checker at the receiver site. The program receives a message of an arbitrary length and a 16-bit checksum calculated at the sender site. It then calculates a new checksum out of both. If the new checksum is 0, it prints a message that the data is valid; otherwise, it prints a message that the data is corrupted.

    a. First test your program with the string and checksum calculated in Program 14-5. The new checksum should be 0.

    b. Then change one of the characters in the string and test the program again. The result should be non-zero.

    c. Note that the checksum itself can be corrupted during the transmission. Also check your program by changing one of the hexadecimal digits of the checksum calculated at the sender site. The result should be non-zero again.

45. There is one major problem with the traditional checksum calculation discussed in the text. If two 16-bit items are transposed in transmission, the checksum cannot catch this error. The reason is that the traditional checksum is not weighted; it treats each data item equally. In other words, the order of data item is immaterial to the calculation.

    To catch transposition errors, J. Fletcher created a new checksum algorithm. The Fletcher checksum was devised to weigh each data item according to its position. The algorithm uses eight bits (known as octets in telecommunication systems) and creates a 16-bit checksum.

The calculation uses modulo 256 (2^8), which means the intermediate results are divided by 256 and the remainder is kept. The algorithm uses two accumulators, `a` and `b`. The first is the sum of the data; the second is a weighted sum. These two accumulators are then used to form the checksum, `a` being the first byte and `b` being the second byte. Figure 14-13 shows the logic diagram for the Fletcher checksum.

The following equations represent the calculations shown in the flowchart.

```
// Calculations in modulo 256
$a = d_1 + d_2 + \cdots + d_n$
$b = nd_1 + (n - 1)d_2 + \cdots + d_n$
```

If, for example, d_1 and d_2 are swapped during the transmission, the calculation of b at the receiver is different from the one done at the sender. Write a function to calculate a Fletcher checksum. Test it with an appropriate test driver.

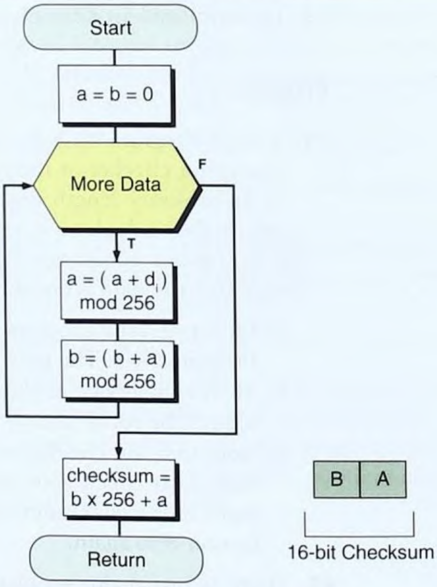


FIGURE 14-13 Fletcher Checksum Design

46. The Adler checksum, designed by Mark Adler, is a 32-bit checksum. It is similar to the Fletcher checksum with three differences. First, calculation is done on two bytes at a time to produce a 32-bit checksum. Second, a is initialized to 1 instead of 0. Third, the modulus is a prime number, 65,521, which is the largest prime number less than 65,536 (2^{32}). It has been

proved that a prime modulo has a better detecting capability in some combinations of data. Figure 14-14 shows a simple, though inefficient, algorithm in flowchart form.

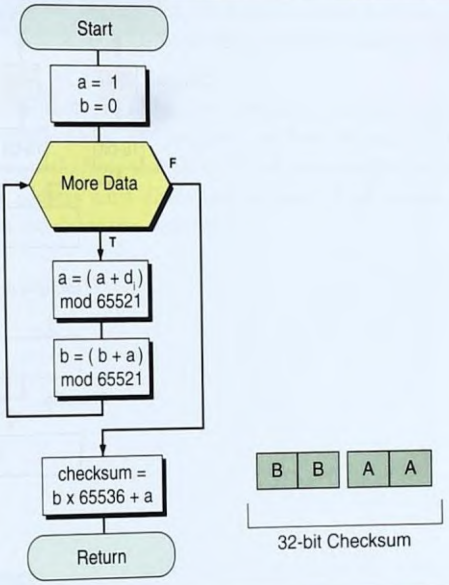


FIGURE 14-14 Adler Checksum Process

It can be proved that the accumulator *B* is the weighted sum of data items. We have

// Calculations in modulo 65521

$a = d_1 + d_2 + \cdot \cdot \cdot + d_n + 1$

$b = nd_1 + (n - 1)d_2 + ... + d_n + n$

Write a program to calculate an Adler checksum.

47. In security, we need to create a digest of a file or a message to protect the message from malicious changes during transmission. A sender can send the message with the digest to a receiver. The receiver then uses the same process and the key to create a new digest. If the received digest is exactly the same as the sent digest, the message has not been changed because the digests are one-to-one functions; probability that two messages create the same digest is extremely low. The secret key between the sender and the receiver does not allow an intruder to create a phony message and digest and pretend that is coming from the true sender. The diagram in Figure 14-15 shows an extremely simplified version of a process that creates a digest.

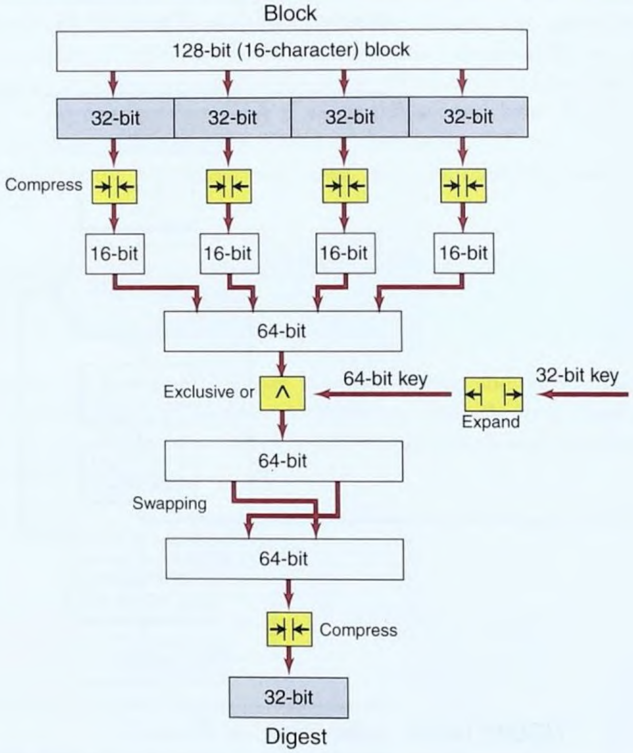


FIGURE 14-15 Simplified Digest Process

a. Each 16-character block is divided into four 32-bit integers.
b. Each 32-bit integer is compressed into a 16-bit integer. The compression removes the first and the last 8 bits.
c. The four 16-bit blocks are combined to make a 64-bit block.
d. A 32-bit key is expanded into a 64-bit key by adding 16 1's at the beginning and 16 1's at the end.
e. The 64-bit key is exclusive *or*'ed with the block from part c.
f. The resulting 64-bit is swapped from the middle. The right 32 bit goes to the left and the right 32 bits goes to the right.
g. The result is compressed to create a 32-bit digest by removing the first and the last 16 bits.

Your program needs to use a main function, two compress functions, an expansions function, a swap function. The main function accepts a 16-character string and one 32-bit key from the keyboard and create and print a 32-bit digest.

48. Combine Programs 14-9 and 14-10 and make them interactive. The new program accepts an address in the form x.y.z.t, and an integer prefix between 0 and 32 and creates the network address (first address), broadcast address (last address), the mask. The program also prints the total number of addresses in the block. This can be done by first changing the first and last address to a 32-bit integer and then finding the difference plus 1.

49. Write a program that, given the first address in a block and the number of addresses in the block, the program can find the last address and the corresponding mask. Note that there is only one restriction to this problem: the first address must divide the number of addresses evenly. Your program must check for this restriction.

Chapter 15

Lists

Aside from some advanced C concepts and functions that we leave for your future studies, we have covered the basics of the C language. In this chapter, we turn our attention to a concept that is pervasive throughout computer science: lists.

A **list** is a collection of related data. We can divide lists into two categories: linear lists and non-linear lists. Linear lists are divided into general lists, stacks, and queues. Non-linear lists are divided into trees and graphs. We define and discuss each of these lists briefly in this chapter. Figure 15-1 contains the breakdown of lists.

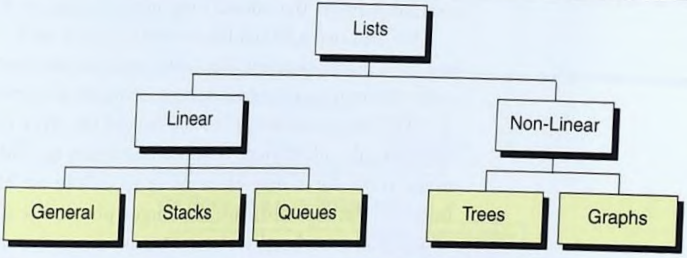


FIGURE 15-1 Lists

Objectives

- □ To introduce the basic concepts of linked lists
- □ To introduce the basic concepts of stacks
- □ To introduce the basic concepts of queues
- □ To introduce the basic concepts of tree structures
- □ To introduce the basic concepts of graph structures

15.1 List Implementations

The C language does not provide any list structures or implementations. When we need them, we must provide the structures and functions for them. Traditionally, two data types, arrays and pointers, are used for their implementation. The array implementation uses static structures that are determined during the compilation or while the program is running. The pointer implementation uses dynamically allocated structures known as linked lists.

Array Implementation

In an array, the sequentiality of a list is maintained by the order structure of elements in the array (indexes). Although searching an array for an individual element can be very efficient, addition and deletion of elements are complex and inefficient processes. For this reason arrays are seldom used, especially when the list changes frequently. In addition, array implementations of lists can become excessively large, especially when there are several successors for each element.

Linked List Implementation

A **linked list** is an ordered collection of data in which each element contains the location of the next element or elements. In a linked list, each element contains two parts: **data** and one or more **links**. The data part holds the application data—the data to be processed. Links are used to chain the data together. They contain pointers that identify the next element or elements in the list.

We can use a linked list to create linear and non-linear structures. In linear lists, each element has only zero or one successor. In non-linear lists, each element can have zero, one, or more successors.

The major advantage of the linked list over the array is that data are easily inserted and deleted. It is not necessary to shift elements of a linked list to make room for a new element or to delete an element. On the other hand, because the elements are no longer physically sequenced, we are limited to sequential searches:[1] we cannot use a binary search. When we examine trees, we discuss the binary search tree, which allows for easy updates and efficient searches.

Figure 15-2(a) shows a linked list implementation of a linear list. The link in each element, except the last, points to its unique successor; the link in the last element contains a null pointer, indicating the end of the list. Figure 15-2(b) shows a linked list implementation of a non-linear list. An element in a non-linear list can have two or more links. Here each element contains two links, each to one successor. Figure 15-2(c) contains an example of an empty list, linear or non-linear. We define an empty list as a pointer with a null value.

In this section, we discuss only the basic concepts for linked lists.

[1] Sequential and binary searches are discussed in Chapter 8.

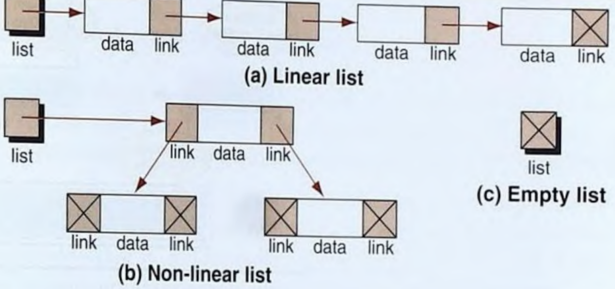


FIGURE 15-2 Linked Lists

Nodes

In linked list implementations, the elements in a list are called nodes. A **node** is a structure that has two parts: the data and one or more links. Figure 15-3 shows two different nodes: one for a linear list and the other for a non-linear list.

The nodes in a linked list are called **self-referential** structures. In a self-referential structure, each instance of the structure contains one or more pointers to other instances of the same structural type. In Figure 15-3, the colored boxes with arrows are the pointers that make the linked list a self-referential structure.

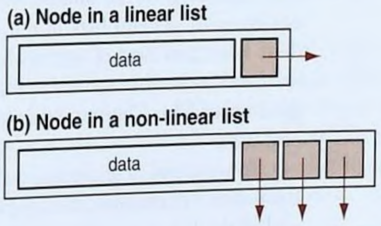


FIGURE 15-3 Nodes

The data part in a node can be a single field, multiple fields, or a structure that contains several fields, but it always acts as a single field. Figure 15-4 shows three designs for a node of a linked list. The upper-left node contains a single field, a number, and a link. The upper-right node is more common. It contains three data fields: a name, an id, and grade points (`grdPts`)—and a link. The third example is the one we recommend. The fields are defined in their own structure, which is then put into the definition of a node structure.

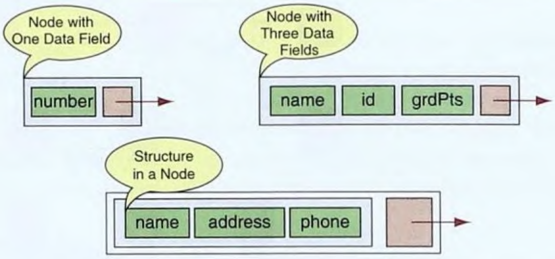


FIGURE 15-4 Linked List Node Structures

Pointers to Linked Lists

A linked list must always have a head pointer. Depending on how we use the list, we may have several other pointers as well. For example, if we are going to search a linear list, we will need an additional pointer to the location where we found the data we were looking for. Furthermore, in many structures, programming is more efficient if there is a pointer to the last node in the list as well as a head pointer.

15.2 General Linear Lists

A general linear list is a list in which operations, such as retrievals, insertions, changes, and deletions, can be done anywhere in the list, that is, at the beginning, in the middle, or at the end of the list. We use many different types of general lists in our daily lives. We use lists of employees, student lists, and lists of our favorite songs. When we process our song list, we need to be able to search for a song, add a new song, or delete one we gave away. This chapter describes how we can maintain and process general lists.

To work with a linear list, we need some basic operations that manipulate the nodes. For example, we need functions to insert a node at the beginning or end of the list, or anywhere in the middle of the list. We also need a function to delete a node from the list. Finally, we need a function to find a requested node in the list. Given these primitive functions, we can build functions that process any linear list.

Insert a Node

We use the following four steps to **insert a node** into a linear list:

1. Allocate memory for the new node.
2. Determine the insertion point—that is, the position within the list where the new data are to be placed. To identify the insertion position, we need to know only the new node's logical predecessor (`pPre`).

3. Point the new node to its successor.

4. Point the predecessor to the new node.

As seen in step 2 above, to insert a node into a list, we need to know the location of the node that precedes the new node (`pPre`). This pointer can be in one of two states: It can contain the address of a node, or it can be null.

If the predecessor is null, then we are inserting to an empty list or at the beginning of the list. If it is not null, then we are inserting somewhere after the first node—that is, in the middle of the list—or at the end of the list. A list and its pointers are shown in Figure 15-5.

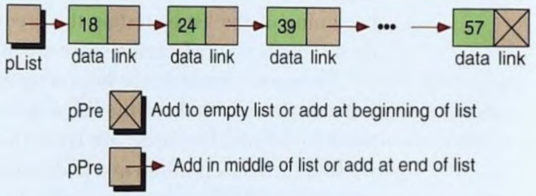
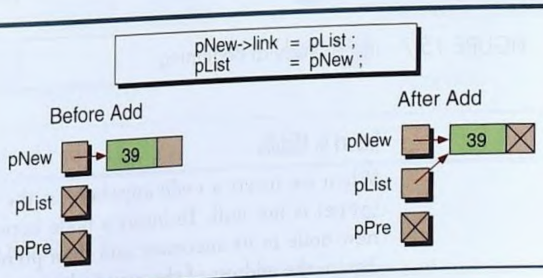

18
24
39
•••
57
pList
data link
data link
data link
data link
pPre
Add to empty list or add at beginning of list
pPre
Add in middle of list or add at end of list

FIGURE 15-5 Pointer Combinations for Insert

We discuss each of these situations in turn.

Insert into Empty List

When the head of the list is null, then the list is empty. This situation is shown in Figure 15-6. All that is necessary to insert a node to an empty list is to point the list head pointer to the address of the new node and make sure that its link field is null. We could use a constant null to set the link field of the new node, but we use the null contained in the list head pointer. The reason for this will become apparent in the next section.



```
pNew->link = pList ;
pList      = pNew ;
```

FIGURE 15-6 Insert Node to Empty List

The statements to insert a node into an empty list are shown below.

```
pNew->link = pList;               // set link to NULL
pList      = pNew;                // point list to first node
```

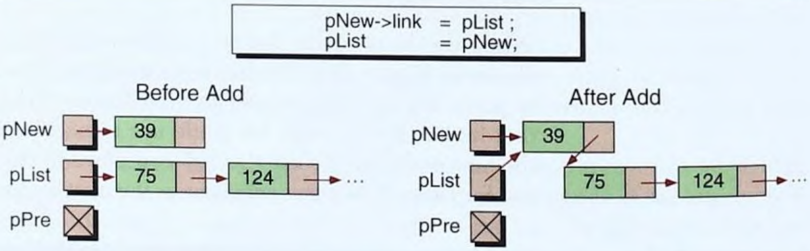
Note the order of these two statements. We must first point the new node to *NULL*, then we can change the head pointer. If we reverse these statements, we end up with the new node pointing to itself, which would put our program into a never-ending loop when we process the list.

Insert at Beginning

We insert a node at the beginning of the list whenever it must be placed before the first node of the list. We determine that we are inserting at the beginning of the list by testing the predecessor pointer (`pPre`). If it is null, then there is no predecessor, so we are at the beginning of the list.

To insert a node at the beginning of the list, we simply point the new node to the first node of the list and then set the head pointer (`pList`) to point to the new first node. We know the address of the new node. How can we find the address of the first node currently in the list so we can point the new node to it? The answer is simple: The first node's address is stored in the head pointer (`pList`).

If you compare these two statements to the statements to insert into an empty list, you will see that they are the same. This is because, logically, inserting into an empty list is the same as inserting at the beginning of a list. We can therefore use the same logic to cover both situations. Inserting at the beginning of the list is shown in Figure 15-7.



```
pNew->link  = pList ;
pList       = pNew;
```

FIGURE 15-7 Insert Node at Beginning

Insert in Middle

When we insert a node anywhere in the middle of the list, the predecessor (`pPre`) is not null. To insert a node between two nodes, we must point the new node to its successor and then point the predecessor to the new node. Again, the address of the new node's successor can be found in the predecessor's link field. The statements to insert a node in the middle of the list are shown in Figure 15-8.

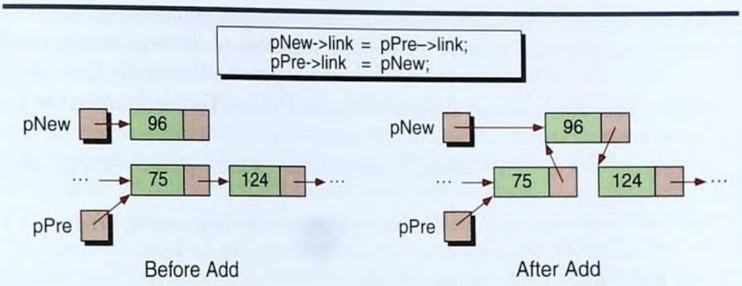


FIGURE 15-8 Insert Node in Middle

Insert at End

When we insert at the end of the list, we only need to point the predecessor to the new node. There is no successor to point to. It is necessary, however, to set the new node's link field to *NULL*. The statements to insert a node at the end of a list are shown below.

```
pNew->link  = NULL;
pPre->link  = pNew;
```

Rather than have special logic in the function for inserting at the end, however, we can take advantage of the existing linear list structure. We know that the last node in the list has a link pointer of *NULL*. If we use this pointer rather than a constant, then the code becomes exactly the same as the code for inserting in the middle of the list. The revised code is shown in Figure 15-9. Compare it to the code in Figure 15-8.

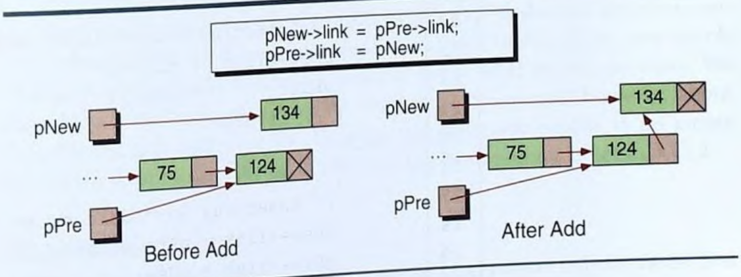


FIGURE 15-9 Insert Node at End

Insert Node Function

Now let's write the function that puts it all together and inserts a node into the list. We are given the head pointer (`pList`), the predecessor (`pPre`), and the data to be inserted (`item`). We must allocate memory for the new node (*malloc*) and adjust the link pointers appropriately. Since we might insert a node before the beginning of the list, we also need to be able to change the contents of the head pointer. We ensure that the head pointer is always correct by returning it. The function declaration for `insertNode` is shown below.

```
NODE* insertNode (NODE* pList, NODE* pPre, DATA item);
```

The complete function is shown in Program 15-1.

PROGRAM 15-1 Insert a Node

```
/* =================== insertNode ===================
   This function inserts a single node into a linear list.
      Pre   pList is pointer to the list; may be null
            pPre points to new node's predecessor
            item contains data to be inserted
      Post  returns the head pointer
*/
NODE* insertNode (NODE* pList, NODE* pPre, DATA item)
{
// Local Declarations
   NODE* pNew;

// Statements
   if (!(pNew  = (NODE*)malloc(sizeof(NODE))))
       printf("\aMemory overflow in insert\n"),
              exit (100);

   pNew->data = item;
   if (pPre == NULL)
      {
       // Inserting before first node or to empty list
       pNew->link  = pList;
       pList       = pNew;
      } // if pPre
   else
      {
       // Inserting in middle or at end
       pNew->link = pPre->link;
       pPre->link = pNew;
      } // else
    return pList;
 } // insertNode
```

Program 15-1 Analysis We have discussed all the logic in this function except the memory allocation. Recall that *malloc* returns either an address in the heap or null when there is no more room in the heap. Therefore, we need to test to see if memory is available.

When memory is exhausted, we have an overflow condition. The action taken depends on the application being programmed; however, the general course of action is to abort the program. We assume the general case in the algorithm; if null is returned from *malloc*, we print a message to the user and exit the program.

With the idea in mind that all code needs to be tested, how can we test the overflow logic? We could use the brute-force method—insert data until the list overflows—but that could take a long time. We tested it by inserting an array of 10,000 long double variables to the data structure. On our system, we got an overflow on the fifth node and the logic was tested.

The technique used to call `insertNode` is important, since it returns the head of the list under all circumstances. The call should assign the return value to the list head pointer. This ensures that when the head of the list changes, the change will be reflected in the calling function. A typical call to insert data into a student list is coded below. Note that the head pointer (`stuList`) is found both in the assignment and as an actual parameter.

```
stuList = insertNode (stuList, pPre, stuData);
```

Delete a Node

When we **delete a node** we logically remove the node from the linear list by changing various link pointers and then physically deleting it from the heap. The delete situations parallel those for insert. We can delete the first node, any node in the middle, or the end node of a list. As you will see below, these three situations reduce to only two combinations: Delete the first node, and delete any other node.

To logically delete a node, we must first locate the node itself (identified by `pCur`) and its predecessor (identified by `pPre`). We discuss location concepts shortly. After we locate the node that we want to delete, we can simply change its predecessor's link field to point to the deleted node's successor. We then recycle the node using *free*. We need to be concerned, however, about deleting the only node in a list. Deleting the only node results in an empty list, so we must be careful that in this case the head pointer is set to *NULL*.

Delete First Node

When we delete the first node, we must reset the head pointer to point to the first node's successor and then recycle (*free*) the deleted note. We can tell we are deleting the first node by testing the predecessor (`pPre`). If the predecessor is null, we are deleting the first node. This situation is diagrammed in Figure 15-10.

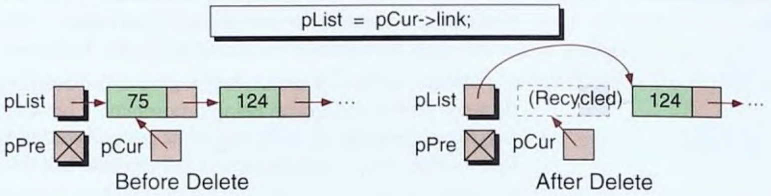


FIGURE 15-10 Delete First Node

The statements to delete the first node are shown below.

```
pList = pCur->link;
free(pCur);
```

If you examine this logic carefully, you will note that it also handles the situation when we are deleting the only node in the list. If the first node is the only node, then its link field is null. Since we move its link field (a null) to the head pointer, the result is by definition an empty list.

General Delete Case

We call deleting any node other than the first a general case, since the same logic handles deleting a node in the middle of the list and deleting a node at the end of the list. For both of these cases, we simply point the predecessor node, identified by `pPre`, to the successor of the node being deleted. The node being deleted is identified by the current node pointer, `pCur`. Its successor is therefore `pCur->link`.

Deleting the last node is handled automatically. When the node being deleted is the last node of the list, its null pointer is moved to the predecessor's link field, making the predecessor the new logical end of the list. After the pointers have been adjusted, the current node is recycled. The delete general case statements are shown below.

```
pPre->link =  pCur->link;
free (pCur);
```

The general case is shown in Figure 15-11.

Function to Delete a Node

The complete logic to delete a node is shown in Program 15-2. It is given a pointer to the head of the list, the node to be deleted, and the delete node's predecessor. After deleting and recycling the node, it returns the pointer to the beginning of the list.

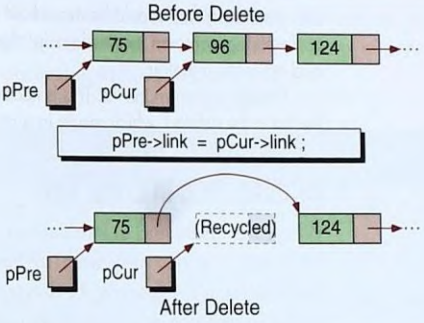


FIGURE 15-11 Delete—General Case

PROGRAM 15-2 Delete a Node

```
/* ==================== deleteNode ====================
   This function deletes a single node from the link list.
      Pre   pList is a pointer to the head of the list
            pPre points to node before the delete node
            pCur points to the node to be deleted
      Post  deletes and recycles pCur
            returns the head pointer
*/
NODE* deleteNode (NODE* pList, NODE* pPre, NODE* pCur)
{
// Statements
   if (pPre == NULL)
      // Deleting first node
      pList = pCur->link;
   else
      // Deleting other nodes
      pPre->link = pCur->link;
   free  (pCur);
   return pList;
}  // deleteNode
```

Program 15-2 Analysis Three points in this function merit discussion. The first and most important is that the node to be deleted must be identified before this function is called. It assumes that the predecessor and current pointers are properly set. If they aren't, then the program most likely fails. Even if it doesn't fail, the data will be wrong. (It is better that the program fail than report invalid results.)

Second, when we discussed the individual logic cases, we placed the recycle statement (*free*) after each delete. In the implementation, we moved it to the end of the function. When the same statements appear in both the true and false blocks of a selection statement, they should be moved out of the selection logic. This is the same concept as factoring common expressions in algebra. The result is a program that is smaller and easier to maintain.

Finally, we return the list. This is necessary because it is possible for the first node of the list to be deleted, which results in a new node being the head of the list.

Locating Data in Linear Lists

When we insert and delete data in a linear list, we must first search the list. To insert a node, we search to identify the logical predecessor of the new node in its key sequence. To delete a node, we search to identify the location of the node to be deleted and its logical predecessor.

Search Linear List

Both the insert and delete for lists require that we search the list. For inserts, we need to know the predecessor to the node to be inserted; for deletes, we need to know the predecessor to the node to be deleted. Also, to locate a node for processing, such as adding to a count or printing its contents, we need to know its location. Although we could write separate search functions for all three cases, the traditional solution is to write one search that satisfies all requirements. This means that our search must return both the predecessor and the current (found) locations.

Generally, lists are maintained in key sequence. A key is a field within a structure that identifies the data. Examples of keys would include student number to identify data about students and part number to identify data about parts. In this text, we limit our discussion of lists to key-sequenced lists, that is, lists in which each key in the list is less than or greater than the key that follows it.

To search a list on a key, we need a key field. For simple lists, the key and the data can be the same field. More complex structures require a separate key field. A generic linked-list node and key structure are seen in the following example.

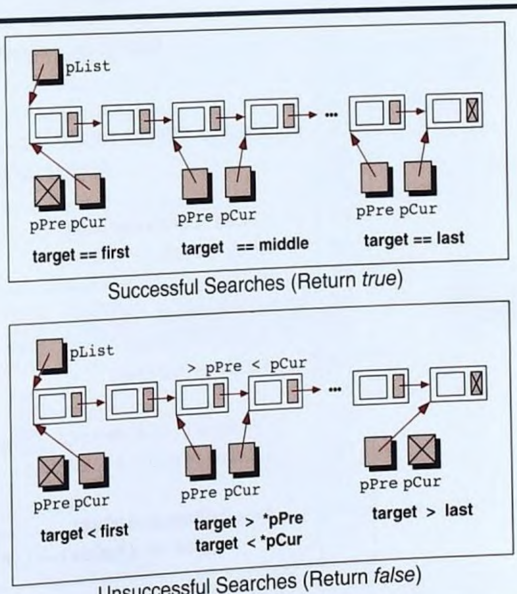
```
// Global Declarations
   typedef int KEY_TYPE;           // Application Dependent

   typedef struct
      {
       KEY_TYPE key;
       …                            // Other Data Fields
      } DATA;

   typedef struct nodeTag
      {
       DATA            data;
       struct nodeTag* link;
      } NODE;
```

Given a target key, the search attempts to locate the requested node in the linear list. If a node in the list matches the target value, the search returns *true*; if no key matches, it returns *false*. The predecessor and current pointers are set according to the rules in Table 15-1.

Condition	pPre	pCur	Return
target < first node	*NULL*	first node	0
target == first node	*NULL*	first node	1
first < target < last	largest node < target	first node > target	0
target == middle node	node's predecessor	equal node	1
target == last node	last's predecessor	last node	1
target > last node	last node	*NULL*	0

TABLE 15-1 Linear List Search Results

Each of these conditions is also shown in Figure 15-12.



Successful Searches (Return *true*)

Unsuccessful Searches (Return *false*)

FIGURE 15-12 Search Results

Since the list is in key sequence, we use a modified version of the sequential search. Knuth[2] calls this search "Sequential search in ordered table." We simply call it an ordered sequential search.

We start at the beginning and search the list sequentially until the target value is no longer greater than the current node's key. At this point, the target value is either less than or equal to the current node's key while the predecessor is pointing to the node immediately before the current node. We now use the current node pointer (pCur) to test for equality and set the return value to *true* if the target value is equal to the list value or *false* if it is less (it cannot be greater) and terminate the search. The code for this search is shown in Program 15-3.

PROGRAM 15-3 Search Linear List

```
/* ==================== searchList ====================
   Given key value, finds the location of a node
      Pre   pList points to a head node
            pPre points to variable to receive pred
            pCur points to variable for current node
            target is key being sought
      Post  pCur points to first node with >= key
            -or- null if target > key of last node
            pPre points to largest node < key
            -or- null if target < key of first node
            function returns true if found
                             false if not found
*/
bool searchList (NODE*  pList, NODE**   pPre,
                 NODE** pCur,  KEY_TYPE target)
{
// Local Declarations
   bool found = false;

// Statements
   *pPre = NULL;
   *pCur = pList;

   // start the search from beginning
   while (*pCur != NULL && target > (*pCur)->data.key)
      {
       *pPre = *pCur;
       *pCur = (*pCur)->link;
```

continued

[2] Donald E. Knuth, *The Art of Computer Programming, Volume 3 Sorting and Searching Second Edition.* (Boston, Mass.: Addison-Wesley, 1979), Algorithm T, p. 398.

PROGRAM 15-3 Search Linear List *(continued)*

```
29          } // while
30
31       if (*pCur && target == (*pCur)->data.key)
32          found = true;
33       return found;
34    }  // searchList
```

Program 15-3 Analysis Examine the *while* statement at statement 25 carefully. Note that there are two tests. The first test protects us from running off the end of the list; the second test stops the loop when we find the target or, if the target doesn't exist, when we find a node larger than the target. It is important to make the null list test first. If the function is at the end of the list, then `pCur` is no longer valid. Testing the key first gives unpredictable results.

We can make the search slightly more efficient if we use a rear pointer. With a rear pointer, we can test the last node before the loop to make sure that the target isn't larger than its key value. If the target is larger, we simply exit the search function after setting the predecessor pointer to the last node and the current pointer to null. Once we know that the target is not greater than the last node, the loop doesn't need to test for running off the end of the list.

Traversing Linear Lists

Algorithms that traverse a list start at the first node and examine each node in succession until the last node has been processed. Several different types of functions use **list traversal** logic. Some examples are printing the list, counting the number of nodes in the list, totalling a numeric field in the node, or calculating the average of a field. In fact, any application that requires processing the entire list uses a traversal. Figure 15-13 is a graphic representation of a linear list traversal.

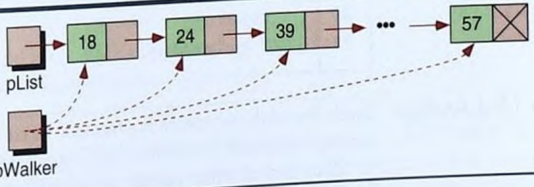


FIGURE 15-13 Linear List Traversal

The basic logic to traverse a linear list is found in the pseudocode shown below. It incorporates two concepts. First, an event loop is used to guard against overrunning the end of the list. Second, after processing the current node, the looping pointer is advanced to the next element.

```
Algorithm traverse list
 1 set pointer to the first node in list
 2 while (not end of the list)
   1  process (current node)
   2  set pointer to next node
 3 end while
end traverse
```

Print Linear List

Program 15-4 uses the traversal logic to print a linear list.

PROGRAM 15-4 Print Linear List

```
 1  /* Traverse and print a linear list.
 2         Pre   pList is a valid linear list
 3         Post  List has been printed
 4  */
 5  void printList (NODE* pList)
 6  {
 7  // Local Declarations
 8     NODE* pWalker;
 9
10  // Statements
11     pWalker = pList;
12     printf("List contains:\n");
13
14     while (pWalker)
15         {
16          printf("%3d ", pWalker->data.key);
17          pWalker = pWalker->link;
18         } // while
19     printf("\n");
20     return;
21  } // printList
```

Program 15-4 Analysis Study the *while* expression in statement 14 carefully. This is the most common form of pointer evaluation for a linear list. C guarantees that the evaluation of a null pointer is *false*, and all other pointer values evaluate to *true*. This expression is therefore equivalent to

```
while (pWalker != NULL)
```

where `pWalker` is a pointer to current node.

The other important piece of code in the traversal is the loop update in statement 17.

```
pWalker = pWalker->link;
```

This statement advances us through the list. If we forget it, we are in a permanent loop. Since pWalker points to the current node, we need to change it to advance to the next node. This is done by simply assigning the link field to pWalker. When we finally arrive at the end of the list, the link is null, which, when assigned to pWalker, terminates the *while* loop.

Linear List Average

Let's write one more linear list traversal algorithm, one that averages the values in a linear list. To pass a linear list to a function, we only need to pass the value of the head pointer. Look at Program 15-5. The actual data structure is not shown, but the function assumes that data.key is an integer field.

PROGRAM 15-5 Average Linear List

```
/* This function averages the values in a linear list.
      Pre   pList is a pointer to a linear list
      Post  list average is returned
*/
double averageList (NODE* pList)
{
// Local Declarations
   NODE* pWalker;
   int   total;
   int   count;

// Statements
   total   = count = 0;
   pWalker = pList;
   while (pWalker)
      {
       total += pWalker->data.key;
       count++;
       pWalker = pWalker->link;
      } // while
   return (double)total / count;
} // averageList
```

Program 15-5 Analysis To traverse the list, we used the descriptive pointer, pWalker, that "walks" us through the list. Although we could have used pList, we advise against ever using the list pointer for any function logic. Keep it in its original state. Then, if you need to modify the function, you will still know its starting point. This is a small point, but it's the small points that make a program easy to maintain.

Building a Linear List

In the previous sections, we wrote three low-level functions for a list—insert, delete, and search—and two applications—print the list and average. It's time to put them all together in a program.

To build the list, we need to get the data, create the node, determine the insertion position in the list, and then insert the new node. The design is shown in Figure 15-14.

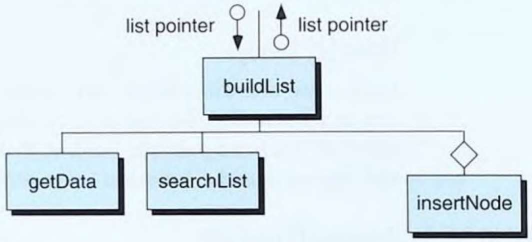


FIGURE 15-14 Design for Inserting a Node in a List

Build List

Building a key-sequenced list uses the search function to determine the insertion position. It uses the design shown in Figure 15-14. The implementation is shown in Program 15-6.

PROGRAM 15-6 Build List

```
/* ==================== buildList ====================
   This program builds a key-sequenced linear list.
      Pre   fileID is file that contains data for list
      Post  list built
            returns pointer to head of list
*/
NODE* buildList (char* fileID)
{
// Local Declarations
   DATA  data;
   NODE* pList;
   NODE* pPre;
   NODE* pCur;
   FILE* fpData;

// Statements
   pList = NULL;

   fpData = fopen(fileID, "r");
   if (!fpData)
      {
       printf("Error opening file %s\a\n", fileID);
```

continued

PROGRAM 15-6 Build List *(continued)*

```
23          exit (210);
24         } // if open fail
25
26     while (getData (fpData, &data))
27         {
28          // Determine insert position
29          searchList (pList, &pPre, &pCur, data.key);
30          pList = insertNode(pList, pPre, data);
31         } // while
32     return pList;
33  } // buildList
```

Program 15-6 Analysis Note that in statement 29, we do not test for duplicate keys when search list returns. If we want to prevent duplicates, we would test the return value and print an error message rather than insert the data when a duplicate key is detected.

Remove a Node

The design to remove a node is also simple. After reading the key to be deleted, we search for its location and its predecessor location and then call the delete node function to do the physical deletion. The design is shown in Figure 15-15.

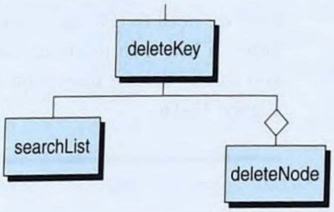


FIGURE 15-15 Design for Remove Node

The code is shown in Program 15-7.

PROGRAM 15-7 Delete Key

```
/* ==================== deleteKey ====================
   Delete node from a linear list.
      Pre   list is a pointer to the head of the list
      Post  node has been deleted
            -or- a warning message printed if not found
            returns pointer to first node (pList)
*/
```

continued

PROGRAM 15-7 Delete Key *(continued)*

```
NODE* deleteKey (NODE* pList)
{
// Local Declarations
   int   key;
   NODE* pHead;
   NODE* pCur;
   NODE* pPre;

// Statements
   printf("Enter key of node to be deleted: ");
   scanf ("%d", &key);

   if (!searchList(pList, &pPre, &pCur, key))
       printf("%d is an invalid key\a\n", key);
   else
       pHead = deleteNode (pList, pPre, pCur);

   return pHead;
} // deleteKey
```

Linear List Test Driver

Now we need to test our functions. To test them, we write a test driver that builds a list, then prints it, inserts more data into it, computes its average, and finally deletes data from it. The design of our test driver is shown in Figure 15-16.

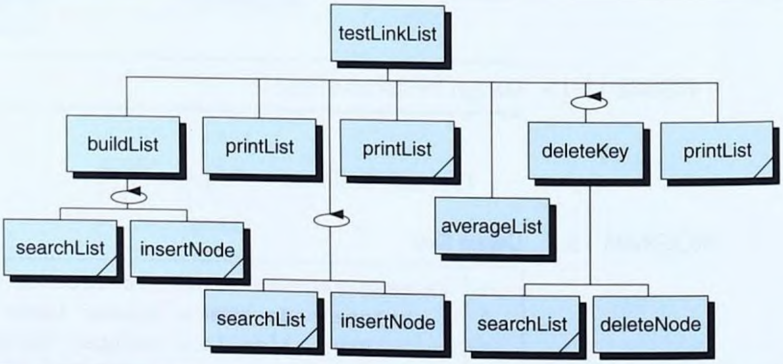


FIGURE 15-16 Link List Test Driver

The code for the test driver is shown in Program 15-8.

PROGRAM 15-8 Test Driver for Link List

```
/* Test driver for list functions.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// Global Declarations
typedef int KEY_TYPE;
typedef struct
   {
    KEY_TYPE key;
   } DATA;
typedef struct nodeTag
   {
    DATA            data;
    struct nodeTag* link;
   } NODE;

// Function Declarations
NODE* insertNode (NODE* pList, NODE* pPre, DATA item);
NODE* deleteNode (NODE* List, NODE* pPre, NODE* pCur);
bool  searchList (NODE* pList, NODE**  pPre,
                  NODE** pCur, KEY_TYPE target);
void  printList  (NODE* pList);
NODE* buildList  (char* fileID);
NODE* deleteKey  (NODE* pList);
bool  getData    (FILE* fpData, DATA* pData);

double averageList (NODE* pList);

int main (void)
{
// Local Declarations
   NODE*  pList;
   NODE*  pPre;
   NODE*  pCur;
   DATA   data;
   double avrg;
   char   more;

// Statements
   printf("Begin list test driver\n\n");
```

continued

PROGRAM 15-8 Test Driver for Link List *(continued)*

```

   // Build List
   pList = buildList("P15-LIST.DAT");
   if (!pList)
      {
       printf("Error building chron file\a\n");
       exit  (100);
      } // if
   printList (pList);

   printf("\nInsert data tests.\n");
   printf("Enter key:                 ");
   scanf ("%d", &data.key);
   do
      {
       if (searchList (pList, &pPre, &pCur, data.key))
          printf("Key already in list. Not inserted\n");
       else
           pList = insertNode(pList, pPre, data);
       printf("Enter key <-1> to stop: ");
       scanf ("%d", &data.key);
      } while (data.key != -1);
   printList (pList);

   avrg = averageList(pList);
   printf("\nData average: %.1f\n", avrg);

   printf("\nDelete data tests.\n");
   do
      {
       pList = deleteKey (pList);
       printf("Delete another <Y/N>: ");
       scanf (" %c", &more);
      } while (more == 'Y' || more == 'y');

   printList (pList);
   printf("\nTests complete.\n");
   return 0;
} // main
```

```
Results:
Begin list test driver

List contains:
```

continued

PROGRAM 15-8 Test Driver for Link List *(continued)*

```
111 222 333 444 555 666 777

Insert data tests.
Enter key:            50
Enter key <-1> to stop: -1
List contains:
 50 111 222 333 444 555 666 777

Data average: 394.8

Delete data tests.
Enter key of node to be deleted: 50
Delete another <Y/N>: n
List contains:
111 222 333 444 555 666 777

Tests complete.
```

Program 15-8 Analysis While the code is rather lengthy, if you have studied each of the functions carefully, you will find this program straightforward. Of greater interest are the test cases needed to completely validate the program. Note that for the inserts, we need to insert at the end, at the beginning, and in the middle. Likewise, for the deletes, we delete the last node, the first node, and one in the middle. We also need to try to delete a node that doesn't exist.

15.3 Stacks

A **stack** is a linear list in which all additions and deletions are restricted to one end, called the **top.** If you insert a data series into a stack and then remove it, the order of the data is reversed. Data input as {5, 10, 15, 20} is removed as {20, 15, 10, 5}. This reversing attribute is why stacks are known as the **last in–first out** (**LIFO**) data structure.

We use many different types of stacks in our daily lives. We often talk of a stack of coins or a stack of dishes. Any situation in which you can only add or remove an object at the top is a stack. If you want to remove any object other than the one at the top, you must first remove all objects above it. A graphic representation of a stack is shown in Figure 15-17.

Although nothing prevents us from designing a data structure that allows us to perform other operations, such as moving the item at the top of the stack to the bottom, the result would not be a stack.

A stack is a last in–first out (LIFO) data structure in which all insertions and deletions are restricted to one end, called the top.

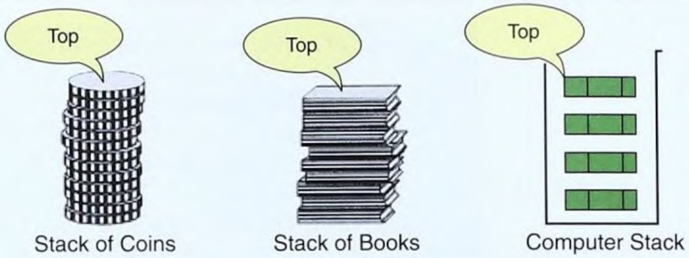


FIGURE 15-17 Stack

Stack Structures

Several data structures can be used to implement a stack. We implement the stack as a linked list.

Data Structure

To implement the linked list stack, we need two different structures, a head node and a data node. The head structure contains **metadata**—that is, data about data—and a pointer to the top of the stack. The data structure contains data and a link pointer to the next node in the stack. The conceptual and physical implementations of the stack are shown in Figure 15-18.

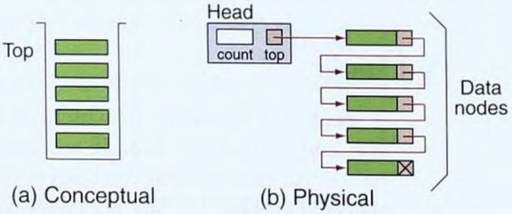
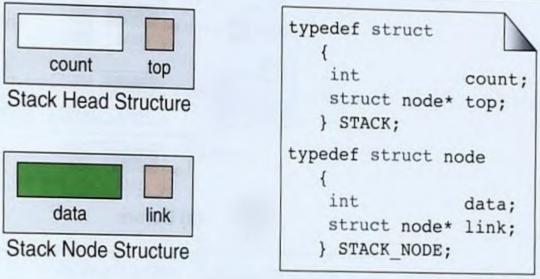


FIGURE 15-18 Conceptual and Physical Stack Implementations

Stack Head

Generally, the **head** for a stack requires only two attributes: a top pointer and a count of the number of elements in the stack. These two elements are placed in a structure. Other stack attributes can be placed here also. A basic head structure is shown in Figure 15-19.



```
typedef struct
   {
    int          count;
    struct node* top;
   } STACK;
typedef struct node
   {
    int          data;
    struct node* link;
   } STACK_NODE;
```

FIGURE 15-19 Stack Data Structure

Stack Data Node

The rest of the data structure is a typical linked list **data node**. Although the application determines the data that are stored in the stack, the stack data node looks like any linked list node. For our example, we use numbers for data. In addition to the data, the data node contains a link pointer to other data nodes, making it a **self-referential structure**. In a self-referential structure, each instance of the structure contains a pointer to another instance of the same structure. The stack data node is also shown in Figure 15-19.

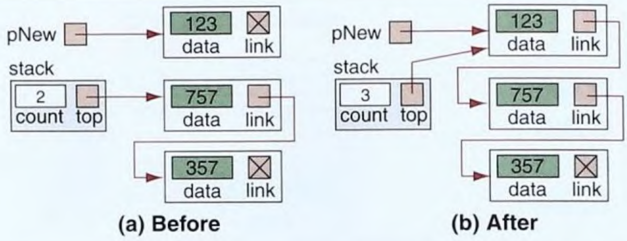
Stack Algorithms

The basic stack operations defined in this section should be sufficient to solve basic stack problems. If an application requires additional stack operations, they can be easily added. For each operation we describe it and then develop it.

Although the implementation of a stack depends somewhat on the implementation language, it is usually implemented with a stack head structure in C. We use the design shown in Figure 15-19.

Push Stack

Push stack inserts an element into the stack. The first thing we need to do when we push data into a stack is find memory for the node. We must therefore allocate a node from dynamic memory. Once the memory is allocated, we simply assign the data to the stack node and then set the link pointer to point to the node currently indicated as the stack top. We also need to update the stack top pointer and add 1 to the stack count field. Figure 15-20 traces a push stack operation in which a new pointer (pNew) is used to identify the data to be inserted into the stack.



(a) Before

(b) After

FIGURE 15-20 Push Stack Example

To develop the insertion algorithm using a linked list, we need to analyze three different stack conditions: (1) insertion into an empty stack, (2) insertion into a stack with data, and (3) insertion into a stack when the available memory is exhausted.

When we insert into a stack that contains data, the new node's link pointer is set to point to the node currently at the top, and the stack's top pointer is set to point to the new node. When we insert into an empty stack, the new node's link pointer is set to null and the stack's top pointer is set to point to the new node. However, because the stack's top pointer is null, we can use it to set the new node's link pointer to null. Thus the logic for inserting into a stack with data and the logic for inserting into an empty stack are identical. Figure 15-20 shows the before and after conditions when we push data into a stack.

As we saw with the linear list, a push can cause an overflow condition. If it does, the function simply returns a false condition. The code is shown in Program 15-9.

PROGRAM 15-9 Push Stack

```
/* =================== push ====================
   Inserts node into linked list stack.
      Pre    pStack is pointer to valid stack header
      Post   dataIn inserted
      Return true  if successful
             false if overflow
*/
bool push (STACK* pStack, int dataIn)
{
// Local Declarations
   STACK_NODE* pNew;
   bool        success;
```

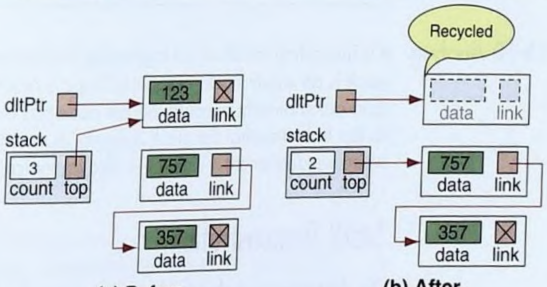
continued

PROGRAM 15-9 Push Stack *(continued)*

```

// Statements
   pNew = (STACK_NODE*)malloc(sizeof (STACK_NODE));
   if (!pNew)
      success = false;
   else
      {
       pNew->data  = dataIn;
       pNew->link  = pStack->top;
       pStack->top = pNew;
       pStack->count++;
       success = true;
      } // else
   return success;
}  // push
```

Pop Stack

Pop stack sends the data in the node at the top of the stack back to the calling algorithm. It then adjusts the pointers to logically delete the node. After the node has been logically deleted, it is physically deleted by recycling the memory, that is, returning it to dynamic memory. After the count is adjusted by subtracting 1, the algorithm returns the status to the caller: if the pop was successful, it returns *true*; if the stack is empty when pop is called, it returns *false*. The operations for pop stack are traced in Figure 15-21.



(a) Before (b) After

FIGURE 15-21 Pop Stack Example

The pop stack code is shown in Program 15-10.

PROGRAM 15-10 Pop Stack

```
/* =================== pop ====================
   Delete node from linked list stack.
      Pre     pStackTop is pointer to valid stack
      Post    dataOut contains deleted data
      Return  true  if successful
              false if underflow
*/
bool pop (STACK* pStack, int* dataOut)
{
// Local Declarations
   STACK_NODE* pDlt;
   bool        success;

// Statements
   if (pStack->top)             // Test for Empty Stack
      {
       success  = true;
       *dataOut = pStack->top->data;
       pDlt     = pStack->top;
       pStack->top = (pStack->top)->link;
       pStack->count--;
       free (pDlt);
      } // else
   else
      success = false;
   return success;
}  // pop
```

Program 15-10 Analysis It is interesting to follow the logic when the last node is being deleted. In this case, the result is an empty stack. No special logic is required; however, the empty stack is created automatically because the last node has a null link pointer, which when moved to top indicates that the stack is empty. A count of zero, which automatically occurs when we decrement the count, is also an indication of an empty stack.

Stack Demonstration

To demonstrate the push and pop stack algorithms, we write a program that inserts random numbers into a stack. After the numbers are inserted, they are popped and printed. When the stack is empty, the program terminates. The design for this program is shown in Figure 15-22.

The node declarations, function declarations, and test driver are contained in Program 15-11.

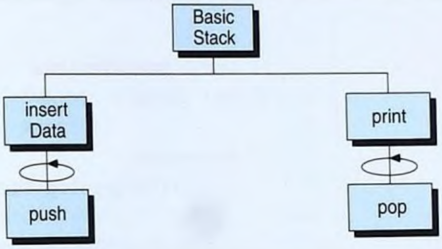


FIGURE 15-22 Design for Basic Stack Program

PROGRAM 15-11 Simple Stack Application Program

```
/* This program is a test driver to demonstrate the
   basic operation of the stack push and pop functions.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// Global Declarations
typedef struct node
   {
    int         data;
    struct node* link;
   } STACK_NODE;

typedef struct
   {
    int         count;
    STACK_NODE* top;
   } STACK;

// Function Declarations
void insertData (STACK* pStack);
void print      (STACK* pStack);
bool push       (STACK* pList, int  dataIn);
bool pop        (STACK* pList, int* dataOut);

int main (void)
```

continued

PROGRAM 15-11 Simple Stack Application Program *(continued)*

```
{
// Local Declarations
STACK* pStack;

// Statements
   printf("Beginning Simple Stack Program\n");

   pStack = malloc(sizeof(STACK));
   if (!pStack)
      printf("Error allocating stack"), exit(100);

   pStack->top   = NULL;
   pStack->count = 0;
   insertData (pStack);
   print      (pStack);

   printf("\nEnd Simple Stack Program\n");
   return 0;
}  // main
```

```
Results:
Beginning Simple Stack Program
Creating numbers: 854 763 123 532  82 632  33 426 228  90
Stack contained:   90 228 426  33 632  82 532 123 763 854
End Simple Stack Program
```

Program 15-11 Analysis

To verify that the program works correctly, we print the numbers as they are generated. This allows us to verify that the stack output was in fact correct. Note that this simple program verifies the LIFO operation of a stack.

Insert Data

The insert data function loops while creating numbers and inserting them into the stack. To create a random number, we use the random number generator and scale the return value to a three-digit range. This code is developed in Program 15-12.

PROGRAM 15-12 Insert Data

```
/* ================== insertData ===================
   This program creates random numbers and
   inserts them into a linked list stack.
      Pre  pStack is a pointer to first node
      Post Stack has been created
*/
void insertData (STACK* pStack)
```

continued

PROGRAM 15-12 Insert Data *(continued)*

```
{
// Local Declarations
   int  numIn;
   bool success;

// Statements
   printf("Creating numbers: ");
   for (int nodeCount = 0; nodeCount < 10; nodeCount++)
      {
       // Generate random number
       numIn  = rand() % 999;
       printf("%4d", numIn);
       success = push(pStack, numIn);
       if (!success)
         {
          printf("Error 101: Out of Memory\n");
          exit (101);
         } // if
      } // for
   printf("\n");
   return;
}  // insertData
```

Print Stack

Once the stack has been built, we print it to verify the output. The print function calls the pop function until the stack is empty. The print code is found in Program 15-13.

PROGRAM 15-13 Print Stack

```
/* ===================== print =====================
   This function prints a singly linked stack.
      Pre     pStack is pointer to valid stack
      Post    data in stack printed
*/
void print (STACK* pStack)
{
// Local Declarations
   int printData;

// Statements
   printf("Stack contained:  ");
   while (pop(pStack, &printData))
      printf("%4d", printData);
```

continued

PROGRAM 15-13 Print Stack *(continued)*

```
    return;
} // print
```

15.4 Queues

A queue is a linear list in which data can be inserted only at one end, called the **rear**, and deleted from the other end, called the **front**. These restrictions ensure that the data are processed through the queue in the order in which they are received. In other words, a queue is a **first in–first out (FIFO)** structure.

A queue is the same as a line. In fact, if you were in England, you would not get into a line, you would get into a queue. A line of people waiting for the bus at a bus station is a queue, a list of calls put on hold to be answered by a telephone operator is a queue, and a list of waiting jobs to be processed by a computer is a queue.

A queue is a linear list in which data can be inserted at one end, called the rear, and deleted from the other end, called the front. It is a first in–first out (FIFO) restricted data structure.

Figure 15-23 shows two representations of a queue: one a queue of people and the other a computer queue. Both people and data enter the queue at the rear and progress through the queue until they arrive at the front. Once they are at the front of the queue, they leave the queue and are served.

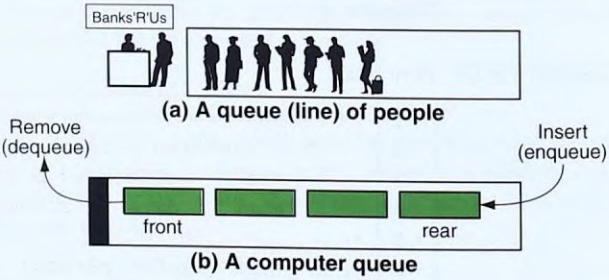


FIGURE 15-23 Queue Concept

Queue Operations

There are two basic queue operations. Data can be inserted at the rear and deleted from the front. Although there are many similarities between stacks and queues, one significant structural difference is that the queue implementation needs to keep track of the front and the rear of the queue, whereas the stack only needs to worry about one end: the top.

Enqueue

The queue insert operation is known as **enqueue**. After the data have been inserted into the queue, the new element becomes the rear. As we saw with stacks, the only potential problem with enqueue is running out of room for the data. If there is not enough room for another element in the queue, the queue is in an overflow state.

Enqueue inserts an element at the rear of the queue.

Figure 15-24 shows the enqueue operation.

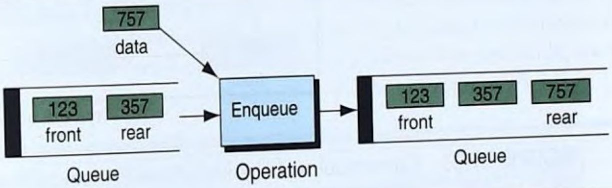


FIGURE 15-24 Enqueue

Dequeue

The queue delete operation is known as **dequeue**. The data at the front of the queue are returned to the user and removed from the queue. If there are no data in the queue when a dequeue is attempted, the queue is in an underflow state.

Dequeue deletes an element at the front of the queue.

The dequeue operation is shown in Figure 15-25.

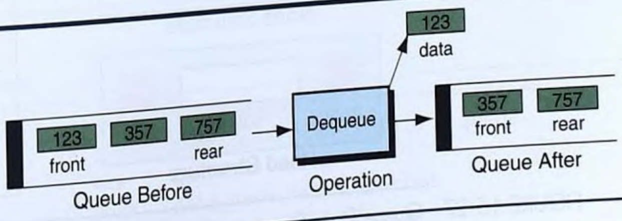


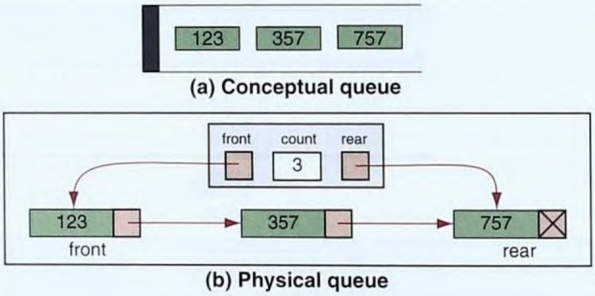
FIGURE 15-25 Dequeue

Queue Linked List Design

As with a stack, we implement our queue using a linked list.

Data Structure

We need two different structures to implement the queue: a queue head structure and a data node structure. After it is created, the queue will have one head node and zero or more data nodes, depending on its current state. Figure 15-26 shows the conceptual and physical implementations for our queue structure.



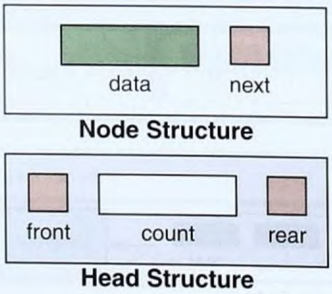
(a) Conceptual queue

(b) Physical queue

FIGURE 15-26 Conceptual and Physical Queue Implementations

Queue Head

The queue requires two pointers and a count. These fields are stored in the **queue head** structure. Other queue attributes, such as the maximum number of items that were ever present in the queue and the total number of items that have been processed through the queue, can be stored in the head node if such data are relevant to an application. The queue head structure is shown in Figure 15-27.



```
typedef struct node
   {
    int            data;
    struct node* next;
   } QUEUE_NODE;

typedef struct
   {
    QUEUE_NODE* front;
    int          count;
    QUEUE_NODE* rear;
   } QUEUE;
```

FIGURE 15-27 Queue Data Structure

Queue Data Node The **queue data node** contains the user data and a link field pointing to the next node, if any. These nodes are stored in dynamic memory and are inserted and deleted as requested by the using program. Its structure is also shown in Figure 15-27.

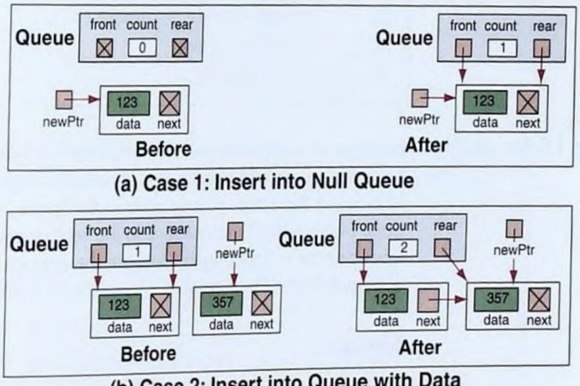
Queue Functions

The two basic functions for a queue are enqueue and dequeue.

Enqueue

The enqueue is a little more complex than inserting data into a stack. To develop the insertion algorithm, we need to analyze two different queue conditions: insertion into an empty queue and insertion into a queue with data. These operations are shown in Figure 15-28.

When we insert data into an empty queue, the queue's front and rear pointers must both be set to point to the new node. When we insert data into a queue with data already in it, we must point both the link field in the last node and the rear pointer to the new node. If the insert was successful, we return a Boolean *true*; if there is no memory left for the new node, we return a Boolean *false*. The code for enqueue is shown in Program 15-14.



(a) Case 1: Insert into Null Queue

(b) Case 2: Insert into Queue with Data

FIGURE 15-28 Enqueue Example

PROGRAM 15-14 Enqueue

```
/* ================= enqueue ================
   This algorithm inserts data into a queue.
      Pre    queue is valid
      Post   data have been inserted
```

continued

PROGRAM 15-14 Enqueue *(continued)*

```
 5          Return true if successful, false if overflow
 6    */
 7    bool enqueue (QUEUE* queue, int dataIn)
 8    {
 9    // Local Declarations
10       QUEUE_NODE* newPtr;
11
12    // Statements
13       if (!(newPtr = malloc(sizeof(QUEUE_NODE))))
14          return false;
15
16       newPtr->data = dataIn;
17       newPtr->next = NULL;
18
19       if (queue->count == 0)
20          // Inserting into null queue
21          queue->front  = newPtr;
22       else
23          queue->rear->next = newPtr;
24       (queue->count)++;
25       queue->rear = newPtr;
26       return true;
27    } // enqueue
```

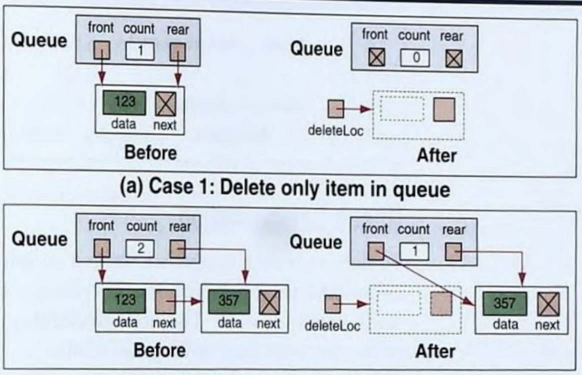
Program 15-14 Analysis Because we must maintain both a front and a rear pointer, we need to check to see if we are inserting into a null queue. If we are, we must set both pointers to the data just inserted. If there are already data in the queue, we need to set the next field of the node at the rear of the queue and the rear pointer to the new node. In this case the front pointer is unchanged. Because the rear pointer is updated in either case, we changed it after the *if* statement (see statement 25).

Dequeue

Although dequeue is also a little more complex than deleting data from a stack, it starts out much the same. We must first ensure that the queue contains data. If the queue is empty, we have underflow and we return *false*, indicating that the dequeue was not successful.

Given that there are data to be dequeued, we pass the data back through the parameter list and then set the front pointer to the next item in the queue. If we have just dequeued the last item, the queue front pointer automatically becomes a null pointer by assigning it the null pointer from the link field of the last node. However, if the queue is now empty, we must also set the rear pointer to null. These cases are shown in Figure 15-29.

The dequeue implementation is shown in Program 15-15.



(a) Case 1: Delete only item in queue

(b) Case 2: Delete item at front of queue

FIGURE 15-29 Dequeue Examples

PROGRAM 15-15 Dequeue

```
/* ================= dequeue ================
   This algorithm deletes a node from the queue.
      Pre    queue is pointer to queue head structure
             dataOut is pointer to data being deleted
      Post   Data pointer to queue front returned and
             front element deleted and recycled.
      Return true if successful; false if underflow
*/
bool dequeue (QUEUE* queue, int* dataOut)
{
// Local Declarations
   QUEUE_NODE* deleteLoc;

// Statements
   if (!queue->count)
       return false;

   *dataOut = queue->front->data;
   deleteLoc = queue->front;
   if (queue->count == 1)
      // Deleting only item in queue
      queue->rear  = queue->front = NULL;
   else
      queue->front = queue->front->next;
   (queue->count)--;
```

continued

PROGRAM 15-15 Dequeue *(continued)*

```
   free (deleteLoc);

   return true;
}  // dequeue
```

Queue Demonstration

To demonstrate a queue, we write a simple program that creates a queue of colors and then prints them. The design closely parallels the design of the stack demonstration. The node declarations, function declarations, and test driver are contained in Program 15-16.

PROGRAM 15-16 Simple Queue Demonstration

```
/* This program is a test driver to demonstrate the
   basic operation of the enqueue and dequeue functions.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

// Global Declarations
typedef struct node
   {
    int          data;
    struct node* next;
   } QUEUE_NODE;

typedef struct
   {
    QUEUE_NODE* front;
    int         count;
    QUEUE_NODE* rear;
   } QUEUE;

// Function Declarations
void insertData (QUEUE* pQueue);
void print      (QUEUE* pQueue);
bool enqueue    (QUEUE* pList, int  dataIn);
bool dequeue    (QUEUE* pList, int* dataOut);

```

continued

PROGRAM 15-16 Simple Queue Demonstration *(continued)*

```
int main (void)
{
// Local Declarations
QUEUE* pQueue;

// Statements
   printf("Beginning Simple Queue Program\n");

   pQueue = malloc(sizeof(QUEUE));
   if (!pQueue)
      printf("Error allocating queue"), exit(100);

   pQueue->front = NULL;
   pQueue->count = 0;
   pQueue->rear  = NULL;

   insertData (pQueue);
   print      (pQueue);

   printf("\nEnd Simple Queue Program\n");
   return 0;
}  // main
```

```
Results:
Beginning Simple Queue Program
Creating numbers:  854 763 123 532  82
Queue contained:   854 763 123 532  82
End Simple Queue Program
```

Insert Data

The insert data function creates a queue by inserting numbers into the queue. The code is developed in Program 15-17.

PROGRAM 15-17 Insert Data

```
/* ================== insertData ==================
   This program creates random number data and
   inserts them into a linked list queue.
      Pre  pQueue is a pointer to first node
      Post Queue created and filled
*/
void insertData (QUEUE* pQueue)
{
// Local Declarations
```

continued

PROGRAM 15-17 Insert Data *(continued)*

```
   int  numIn;
   bool success;

// Statements
   printf("Creating numbers: ");
   for (int nodeCount = 0; nodeCount < 5; nodeCount++)
       {
        // Generate random number
        numIn  = rand() % 999;
        printf("%4d", numIn);
        success = enqueue(pQueue, numIn);
        if (!success)
           {
            printf("Error 101: Out of Memory\n");
            exit (101);
           } // if
       } // for
   printf("\n");
   return;
} // insertData
```

Print Queue

Once the queue has been built, we print it to verify that the queue was built correctly. The print function calls the dequeue function until all items have been printed. The print code is found in Program 15-18.

PROGRAM 15-18 Print Queue

```
/* ===================== print ====================
   This function prints a singly linked queue.
      Pre     pQueue is pointer to valid queue
      Post    data in queue printed
*/
void print (QUEUE* pQueue)
{
// Local Declarations
   int printData;

// Statements
   printf("Queue contained:  ");
   while (dequeue(pQueue, &printData))
       printf("%4d", printData);
   return;
} // print
```

15.5 Trees

The study of trees in mathematics can be traced to Gustav Kirchhoff in the middle nineteenth century and several years later to Arthur Cayley, who used trees to study the structure of algebraic formulas. Cayley's work undoubtedly laid the framework for Grace Hopper's use of trees in 1951 to represent arithmetic expressions. Hopper's work bears a strong resemblance to today's binary tree formats.[3]

Trees are used extensively in computer science to represent algebraic formulas; as an efficient method for searching large, dynamic lists; and for such diverse applications as artificial intelligence systems and encoding algorithms.

Basic Tree Concepts

A **tree** consists of a finite set of elements, called **nodes**, and a finite set of directed lines, called **branches**, that connect the nodes. The number of branches associated with a node is the **degree** of the node. When the branch is directed toward the node, it is an **indegree** branch; when the branch is directed away from the node, it is an **outdegree** branch. The sum of the indegree and outdegree branches is the degree of the node.

A tree consists of a finite set of elements, called nodes, and a finite set of directed lines, called branches, that connect the nodes.

If the tree is not empty, the first node is called the **root**. The indegree of the root is, by definition, zero. With the exception of the root, all of the nodes in a tree must have an indegree of exactly one; that is, they may have only one predecessor. All nodes in the tree can have zero, one, or more branches leaving them; that is, they may have an outdegree of zero, one, or more (zero or more successors). Figure 15-30 is a representation of a tree.

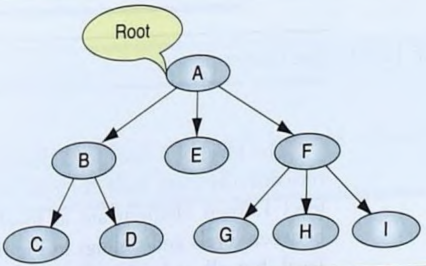


FIGURE 15-30 Tree

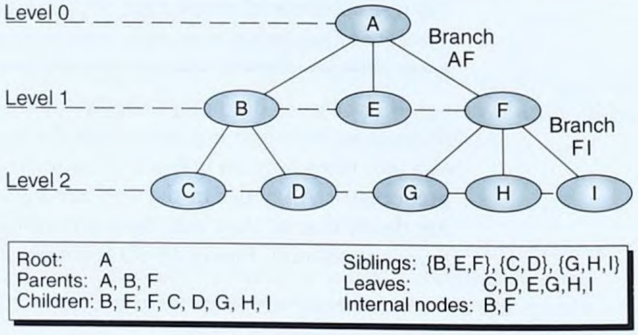
[3] Donald E. Knuth, *op. cit.*, 405, 458.

Terminology

In addition to *root*, many different terms are used to describe the attributes of a tree. A **leaf** is any node with an outdegree of zero, that is, a node with no successors. A node that is not a root or a leaf is known as an **internal node** because it is found in the middle portion of a tree.

A node is a **parent** if it has successor nodes—that is, if it has an outdegree greater than zero. Conversely, a node with a predecessor is a **child**. A child node has an indegree of one. Two or more nodes with the same parent are **siblings**. Fortunately, we don't have to worry about aunts, uncles, nieces, nephews, and cousins. Although some literature uses the term *grandparent*, we do not. We prefer the more general term *ancestor*. An **ancestor** is any node in the path from the root to the node. A **descendent** is any node in the path below the parent node; that is, all nodes in the paths from a given node to a leaf are descendents of that node. Figure 15-31 shows the usage of these terms.

Several terms drawn from mathematics or created by computer scientists are used to describe attributes of trees and their nodes. A **path** is a sequence of nodes in which each node is adjacent to the next one. Every node in the tree can be reached by following a unique path starting from the root. In Figure 15-31 the path from the root to the leaf `I` is designated as `AFI`. It includes two distinct branches, `AF` and `FI`.



Root: A
Parents: A, B, F
Children: B, E, F, C, D, G, H, I

Siblings: {B,E,F}, {C,D}, {G,H,I}
Leaves: C,D, E,G,H,I
Internal nodes: B, F

FIGURE 15-31 Tree Nomenclature

The **level** of a node is its distance from the root. Because the root has a zero distance from itself, the root is at level 0. The children of the root are at level 1, their children are at level 2, and so forth. Note the relationship between levels and siblings in Figure 15-31. Siblings are always at the same level, but all nodes in a level are not necessarily siblings. For example, at level 2, `C` and `D` are siblings, as are `G`, `H`, and `I`. However, `D` and `G` are not siblings because they have different parents.

The **height** of the tree is the level of the leaf in the longest path from the root plus 1. By definition the height of an empty tree is `-1`. Figure 15-31 contains nodes at three levels: 0, 1, and 2. Its height is 3. Because the tree is drawn upside down, some texts refer to the **depth** of a tree rather than its height.

The level of a node is its distance from the root. The height of a tree is the level of the leaf in the longest path from the root plus 1.

A tree may be divided into subtrees. A **subtree** is any connected structure below the root. The first node in a subtree is known as the root of the subtree and is used to name the subtree. Subtrees can also be further subdivided into subtrees. In Figure 15-32, `BCD` is a subtree, as are `E` and `FGHI`. Note that by this definition, a single node is a subtree. Thus, the subtree `B` can be divided into two subtrees, `C` and `D`, and the subtree `F` contains the subtrees `G`, `H`, and `I`.

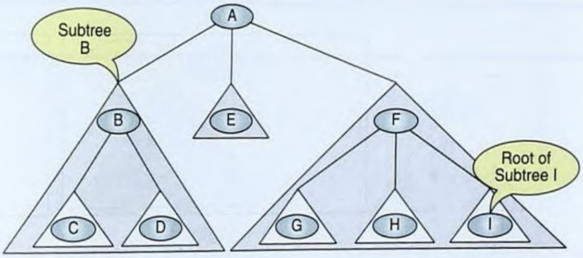


FIGURE 15-32 Subtrees

The concept of subtrees leads us to a recursive definition of a tree: A tree is a set of nodes that either: (1) is empty or (2) has a designated node, called the root, from which hierarchically descend zero or more subtrees, which are also trees.

A tree is a set of nodes that either:

1. **Is empty, or**
2. **Has a designated node, called the root, from which hierarchically descend zero or more subtrees, which are also trees**

Binary Trees

A **binary tree** is a tree in which no node can have more than two subtrees; the maximum outdegree for a node is two. In other words, a node can have zero, one, or two subtrees. These subtrees are designated as the **left subtree** and the **right subtree**.

Figure 15-33 contains a collection of eight binary trees, the first of which is a **null tree**, that is, a tree with no nodes. As you study this figure, note that symmetry is not a tree requirement.

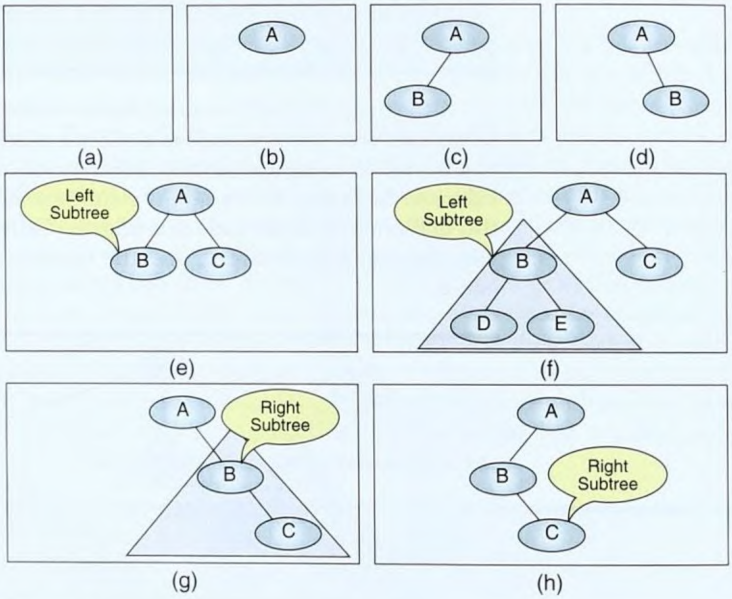


FIGURE 15-33 Collection of Binary Trees

Binary Tree Data Structure

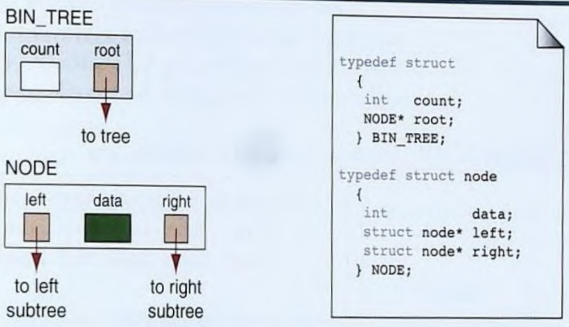
A binary tree needs two separate data structures: one for the head and one for the nodes. As with the linear list, the structure uses a simple head structure that contains a count and a root pointer.

The binary tree nodes contain the application data and two self-referential pointers to the left and right subtrees. These data structures are shown in Figure 15-34.

Binary Tree Traversals

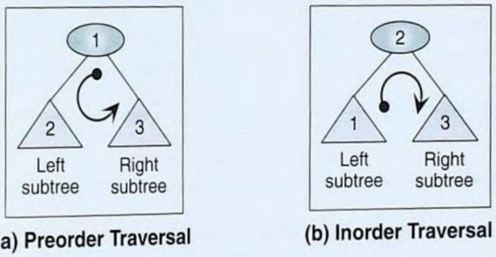
Given that a binary tree consists of a root, a left subtree, and a right subtree, we can define six different traversal sequences. Computer scientists have assigned three of these sequences standard names in the literature; the other three are unnamed but are easily derived. The first two standard traversals are shown in Figure 15-35. Note that we only discuss the preorder and inorder traversals in this text. The others will be covered in your data structures class.

The traditional terminology for the traversals uses a designation of `node` (N) for the root, `left` (L) for the left subtree, and `right` (R) for the right subtree. To demonstrate the different traversal sequences for a binary tree, we use Figure 15-36.



```
typedef struct
  {
   int   count;
   NODE* root;
  } BIN_TREE;

typedef struct node
  {
   int          data;
   struct node* left;
   struct node* right;
  } NODE;
```

FIGURE 15-34 Binary Tree Data Structure



(a) Preorder Traversal

(b) Inorder Traversal

FIGURE 15-35 Binary Tree Traversals

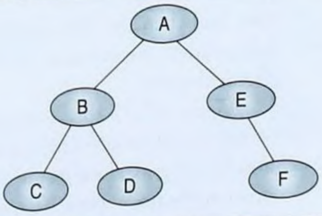


FIGURE 15-36 Binary Tree for Traversals

Preorder Traversal (NLR)

In the **preorder traversal**, the root node is processed first, followed by the left subtree and then the right subtree. It draws its name from the Latin prefix *pre*, which means to go before. Thus, the root goes before the subtrees.

In the preorder traversal, the root is processed first, before its subtrees.

Given the recursive characteristics of trees, it is only natural to implement tree traversals recursively. First we process the root, then the left subtree, and then the right subtree. The left subtree is in turn processed recursively, as is the right subtree. The code for the preorder traversal is shown in Program 15-19.

PROGRAM 15-19 Preorder Traversal of a Binary Tree

```
/* Traverse a binary tree and print its data (integers)
      Pre  root is entry node of a tree or subtree
      Post each node has been printed
*/
void preOrder (NODE* root)
{
// Statements
   if (root)
      {
       printf("%4d", root->data);
       preOrder (root->left);
       preOrder (root->right);
      } // if
   return;
}  // preOrder
```

Using the tree in Figure 15-36 the processing sequence for a preorder traversal processes this tree as follows: First we process the root A. After the root, we process the left subtree. To process the left subtree, we first process its root, B, then its left subtree and right subtree in order. When B's left and right subtrees have been processed in order, we are then ready to process A's right subtree, E. To process the subtree E, we first process the root and then the left subtree and the right subtree. Because there is no left subtree, we continue immediately with the right subtree, which completes the tree.

Figure 15-37 shows another way to visualize the traversal of the tree. Imagine that we are walking around the tree, starting on the left of the root and keeping as close to the nodes as possible. In the preorder traversal we process the node when we meet it for the first time (on the left of the node). This is shown as a black box on the left of the node. The path is shown as a line following a route completely around the tree and back to the root.

Figure 15-38 shows the recursive algorithmic traversal of the tree. The first call processes the root of the tree, A. It then recursively calls itself to process the root of the subtree B, as shown in Figure 15-38(b). The third call, shown in Figure 15-38(c), processes node C, which is also subtree C. At this point we call preorder with a null pointer, which results in an immediate return to subtree C to process its right subtree. Because C's right subtree is also null, we return to node B so that we can process its right tree, D, in Figure 15-38(d). After processing node D, we make two more calls, one with D's null left pointer and one with its null right pointer. Because subtree B has now been completely processed, we return to the tree root and process its right subtree, E, in Figure 15-38(e). After a call to E's null left subtree, we call E's right subtree, F, in Figure 15-38(f). Although the tree is completely processed at this point, we still have two more calls to make: one to F's null left subtree and one to its null right subtree. We can now back out of the tree, returning first to E and then to A, which concludes the traversal of the tree.

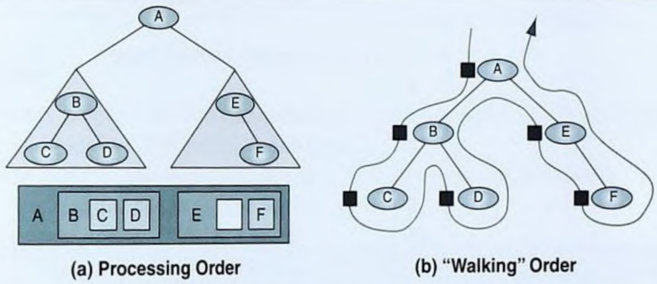


FIGURE 15-37 Preorder Traversal—A B C D E F

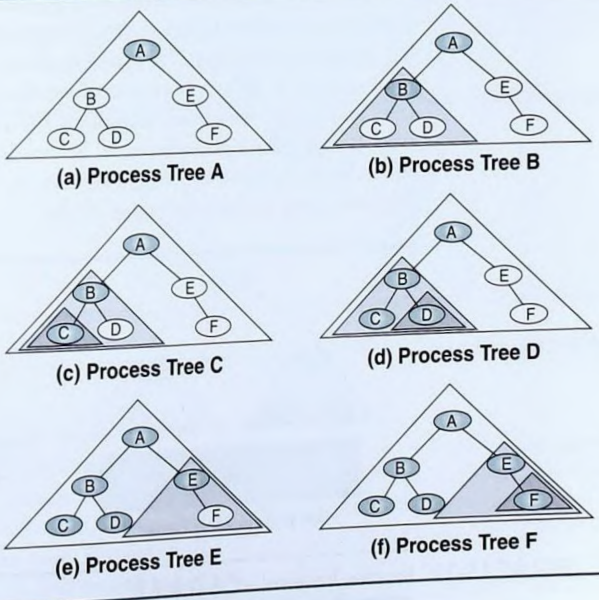


FIGURE 15-38 Algorithmic Traversal of Binary Tree

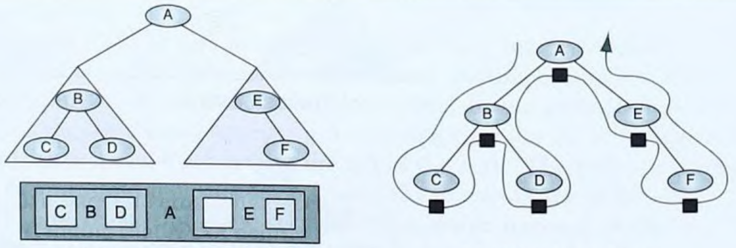
Inorder Traversal (LNR)

The **inorder traversal** processes the left subtree first, then the root, and finally the right subtree. The meaning of the prefix *in* is that the root is processed *in between* the subtrees. Once again we implement the algorithm recursively, as shown in Program 15-20.

PROGRAM 15-20 Inorder Traversal of a Binary Tree

```
/* Traverse a binary tree and print its data (integers)
      Pre  root is entry node of a tree or subtree
      Post each node has been printed
*/
void inOrder (NODE* root)
{
// Statements
   if (root)
      {
       inOrder (root->left);
       printf("%4d", root->data);
       inOrder (root->right);
      } // if
   return;
}  // inOrder
```

Because the left subtree must be processed first, we trace from the root to the far-left leaf node before processing any nodes. After processing the far-left subtree, C, we process its parent node, B. We are now ready to process the right subtree, D. Processing D completes the processing of the root's left subtree, and we are now ready to process the root, A, followed by its right subtree. Because the right subtree, E, has no left child, we can process its root immediately followed by its right subtree, F. The complete sequence for inorder processing is shown in Figure 15-39.



(a) Processing Order

(b) "Walking" Order

FIGURE 15-39 Inorder Traversal—C B D A E F

To walk around the tree in inorder sequence, we follow the same path but process each node when we meet it for the second time (the bottom of the node). This processing route is shown in Figure 15-39(b).

In the inorder traversal, the root is processed between its subtrees.

Binary Search Trees

In this section we define and discuss one of the most common binary trees, the binary search trees. The binary search tree is constructed so that when the tree is traversed using an inorder traversal, the data are in ascending sequence.

Definition

A **binary search tree** (**BST**) is a binary tree with the following properties:

- All items in the left subtree are less than the root.
- All items in the right subtree are greater than or equal to the root.
- Each subtree is itself a binary search tree.

In a binary search tree, the left subtree contains key values less than the root, and the right subtree contains key values greater than or equal to the root.

Generally, the information represented by each node is a record rather than a single data element. When the binary search tree definition is applied to a record, the sequencing properties refer to the key of the record. Figure 15-40 reflects the properties of a binary tree in which K is the key.

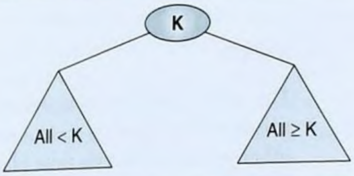


FIGURE 15-40 Binary Search Tree

Figure 15-41 contains five binary search trees.

Now let's look at some binary trees that do not have the properties of a binary search tree. Examine the binary trees in Figure 15-42. The first tree, Figure 15-42(a), breaks the first rule: all items in the left subtree must be less than the root. The key in the left subtree (22) is greater than the key in the root (17). The second tree, Figure 15-42(b), breaks the second rule: all items in the right subtree must be greater than or equal to the root. The key in the right subtree (11) is less than the key in the root (17). Figure 15-42(c) breaks the third rule: each subtree must be a binary search tree. In this tree the left subtree key (6) is less than the root (17), and the right subtree key (19) is greater than the root. However, the left subtree is not a valid binary search tree because it breaks the first rule: its left subtree (11) is greater than the root (6). Figure 15-42(d) also breaks one of the three rules. Do you see which one? (*Hint:* What is the largest key in the left subtree?)

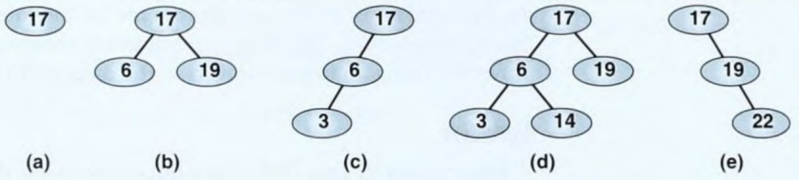


FIGURE 15-41 Valid Binary Search Trees

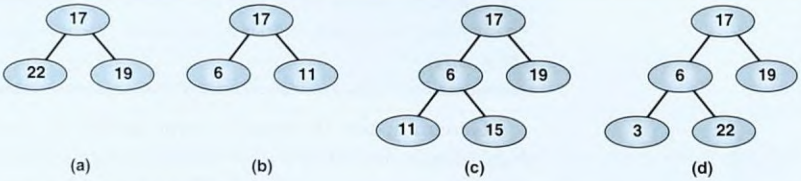


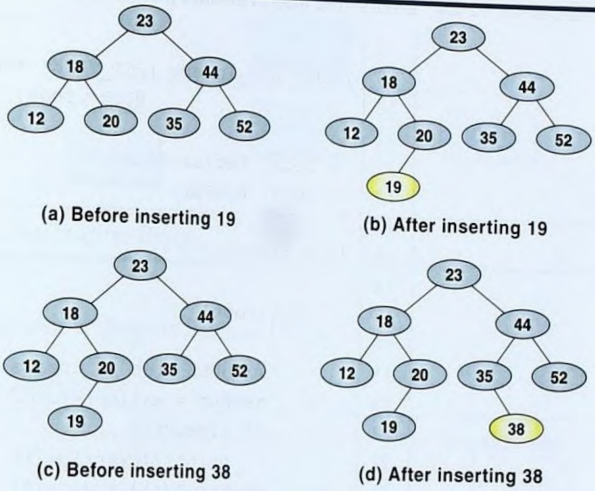
FIGURE 15-42 Invalid Binary Search Trees

Insertion

The **insert node** function adds data to a BST. To insert data all we need to do is follow the branches to an empty subtree and then insert the new node. In other words, all inserts take place at a leaf or at a leaflike node—a node that has at least one null subtree.

All BST insertions take place at a leaf or a leaflike node.

Figure 15-43 shows our binary search tree after we have inserted two nodes. We first added node 19. To locate its insertion point, we searched the tree through the path 23, 18, and 20 to a null left branch. After locating the insertion point, we inserted the new node as the left subtree of 20. We then added 38. This time we searched the tree through 23, 44, and 35 to a null right subtree and inserted the new node.



(a) Before inserting 19

(b) After inserting 19

(c) Before inserting 38

(d) After inserting 38

FIGURE 15-43 BST Insertion

Insertions of both 19 and 38 were made at a leaf node. If we inserted a duplicate of the root, 23, it would become the left subtree of 35. Remember that in a binary search tree, nodes with equal values are inserted in the right subtree. The path for its insertion would therefore be 23, 44, and 35. In this case the insertion takes place at a leaflike node. Although 35 has a right subtree, its left subtree is null. We would therefore place the new node, 23, as the left subtree of 35.

We are now ready to develop the insert algorithm. We can write an elegant algorithm that inserts the data into the tree using recursion. If the tree or subtree is empty, we simply insert the data at the root. If we are not at an empty tree, we determine which branch we need to follow and call recursively to determine whether we are at a leaf yet. The code is shown in Program 15-21. Note that as we saw with other dynamic structures, the BST insert can raise an overflow error.

PROGRAM 15-21 Binary Tree Insert Function

```
/* ==================== BST_Insert ====================
   This function uses recursion to insert the new data
   into a leaf node in the BST tree.
      Pre    Application has called BST_Insert, which
             passes root and data pointer
      Post   Data have been inserted
      Return pointer to [potentially] new root
```

continued

PROGRAM 15-21 Binary Tree Insert Function *(continued)*

```
 8 */
 9 NODE* BST_Insert (BST_TREE* tree,
10                   NODE* root,     int dataIn)
11 {
12 // Local Declarations
13    NODE* newPtr;
14
15 // Statements
16
17    if (!root)
18       {
19        // NULL tree -- create new node
20        newPtr = malloc(sizeof (NODE));
21        if (!newPtr)
22           printf("Overflow in Insert\n"), exit (100);
23        newPtr->data = dataIn;
24        newPtr->left = newPtr->right = NULL;
25        return newPtr;
26       } // if
27
28    // Locate null subtree for insertion
29    if (dataIn < root->data)
30        root->left = BST_Insert(tree, root->left,
31                                dataIn);
32    else
33       // new data >= root data
34        root->right = BST_Insert(tree, root->right,
35                                 dataIn);
36    return root;
37 }  // BST_Insert
```

Program 15-21 Analysis This algorithm must be carefully studied to fully understand its logic. It begins with a recursive search to locate the correct insertion point in a leaf node. A leaf node is identified by a subtree pointer, either right or left, that is null. When we find a leaf pointer, we create a new node and return its address so that it can be inserted into the parent pointer (statement 25).

Because this is a recursive function, it must have a base case. Can you see it? The base case occurs when we locate a leaf and return `newPtr` in statement 25. At this point we begin to back out of the tree.

Binary Tree Example

Let's write a program that builds a binary search tree. We build the tree by asking the user to enter numbers. Once the tree is built, we print it in both preorder and inorder sequence to verify the insertions. To keep the program simple, we will use integer data. The design is shown in Figure 15-44.

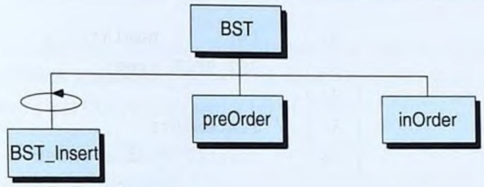


FIGURE 15-44 Binary Tree Program Design

The code is shown in Program 15-22.

PROGRAM 15-22 Binary Tree Example

```
/* Demonstrate the binary search tree insert and
   traversals.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>

// Global Declarations
typedef struct node
   {
    int          data;
    struct node* left;
    struct node* right;
   } NODE;

typedef struct
   {
    int   count;
    NODE* root;
   } BST_TREE;

// Function Declarations
void  preOrder   (NODE* root);
void  inOrder    (NODE* root);
NODE* BST_Insert (BST_TREE* tree,
                  NODE* root,     int data);


int main (void)
{
// Local Declarations
```

continued

PROGRAM 15-22 Binary Tree Example *(continued)*

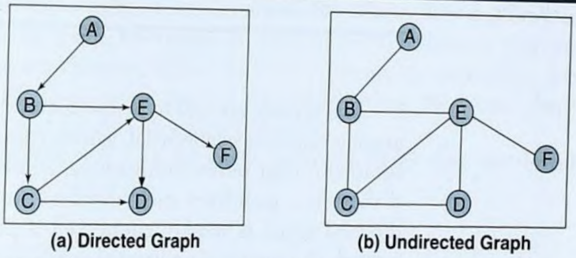
```
   int      numIn;
   BST_TREE tree;

// Statements
   printf("Please enter a series of integers."
          "\nEnter a negative number to stop\n");

   tree.count = 0;
   tree.root  = NULL;
   do
      {
       printf("Enter a number: ");
       scanf("%d", &numIn);
       if (numIn > 0)
          {
           tree.root = BST_Insert
               (&tree, tree.root, numIn);
           tree.count++;
          } // if
      } while (numIn > 0);

   printf("\nData in preOrder: ");
   preOrder (tree.root);

   printf("\n\nData in inOrder:  ");
   inOrder (tree.root);

   printf("\n\nEnd of BST Demonstration\n");
   return 0;
}  // main
```

```
Results
Please enter a series of integers.
Enter a negative number to stop
Enter a number: 45
Enter a number: 54
Enter a number: 23
Enter a number: 32
Enter a number: 3
Enter a number: -1

Data in preOrder:   45  23   3  32  54

Data in inOrder:     3  23  32  45  54

End of BST Demonstration
```

15.6 Graphs

A **graph** is a collection of nodes, called **vertices**, and a collection of segments, called **lines**, connecting pairs of vertices. In other words, a graph consists of two sets, a set of vertices and a set of lines.

Graphs may be either directed or undirected. A **directed graph**, or **digraph** for short, is a graph in which each line has a direction (arrow head) to its successor. The lines in a directed graph are known as **arcs**. In a directed graph, the flow along the arcs between two vertices can follow only the indicated direction. An **undirected graph** is a graph in which there is no direction (arrow head) on any of the lines, which are known as **edges**. In an undirected graph, the flow between two vertices can go in either direction. Figure 15-45 contains an example of both a directed graph (a) and an undirected graph (b).



(a) Directed Graph

(b) Undirected Graph

FIGURE 15-45 Directed and Undirected Graphs

A **path** is a sequence of vertices in which each vertex is adjacent to the next one. In Figure 15-45, {A, B, C, E} is one path and {A, B, E, F} is another. Note that both directed and undirected graphs have paths. In an undirected graph, you may travel in either direction.

Graphs may be directed or undirected. In a directed graph, each line, called an arc, has a direction indicating how it may be traversed. In an undirected graph, the line is known as an edge, and it may be traversed in either direction.

Two vertices in a graph are said to be **adjacent vertices** (or neighbors) if there is a path of length 1 connecting them. In Figure 15-45(a), B is adjacent to A, whereas E is not adjacent to D; on the other hand, D is adjacent to E. In Figure 15-45(b), E and D are adjacent, but D and F are not.

A graph is a collection of nodes, called vertices, and line segments, called arcs or edges, that connect pairs of nodes.

A **cycle** is a path consisting of at least three vertices that starts and ends with the same vertex. In Figure 15-45(b), B, C, D, E, B is a cycle. Note, however, that the same vertices in Figure 15-45(a) do not constitute a cycle because in a digraph a path can follow only the direction of the arc, whereas in an undirected graph a path can move in either direction along the edge. A **loop** is a special case of a cycle in which a single arc begins and ends with the same vertex. In a loop the end points of the line are the same. Figure 15-46 contains a loop.

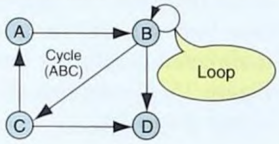
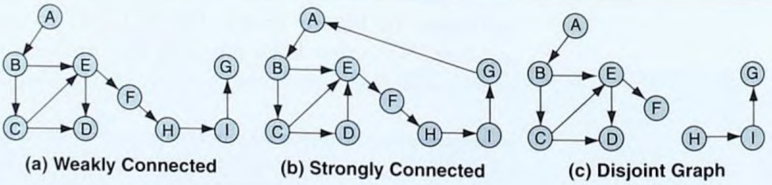


FIGURE 15-46 Cycles and Loops

Two vertices are said to be **connected** if there is a path between them. A graph is said to be connected if, ignoring direction, there is a path from any vertex to any other vertex. Furthermore, a directed graph is **strongly connected** if there is a path from each vertex to every other vertex in the digraph. A directed graph is **weakly connected** if at least two vertices are not connected. (A connected undirected graph would always be strongly connected, so the concept is not normally used with undirected graphs.) A graph is a **disjoint graph** if it is not connected. Figure 15-47 contains a weakly connected graph (a), a strongly connected graph (b), and a disjoint graph (c).



(a) Weakly Connected

(b) Strongly Connected

(c) Disjoint Graph

FIGURE 15-47 Connected and Disjoint Graphs

The **degree** of a vertex is the number of lines incident to it. In Figure 15-47(a) the degree of vertex B is 3 and the degree of vertex E is 4. The **outdegree** of a vertex in a digraph is the number of arcs leaving the vertex; the **indegree** is the number of arcs entering the vertex. Again, in Figure 15-47(a) the indegree of vertex B is 1 and its outdegree is 2; in Figure 15-47(b) the indegree of vertex E is 3 and its outdegree is 1.

One final point: a tree is a graph in which each vertex has only one predecessor; however, a graph is not a tree. We will see later in the chapter that some graphs have one or more trees in them that can be algorithmically determined.

Graph Traversal

A complete discussion of graph algorithms is beyond the scope of this text. As an example of a graph algorithm, we discuss two graph traversals. The remaining algorithms to maintain and traverse graphs are left for your data structures course.

There is always at least one application that requires that all vertices in a given graph be visited; that is, there is at least one application that requires that the graph be traversed. Because a vertex in a graph can have multiple parents, the traversal of a graph presents some problems not found in the traversal of linear lists and trees. Specifically, we must somehow ensure that we process the data in each vertex only once. However, because there are multiple paths to a vertex, we may arrive at it from more than one direction as we traverse the graph. The traditional solution to this problem is to include a visited flag at each vertex. Before the traversal we set the visited flag in each vertex to *off*. Then, as we traverse the graph, we set the visited flag to *on* to indicate that the data have been processed.

There are two standard graph traversals: depth first and breadth first. Both use the visited flag.

Depth-first Traversal

In the **depth-first traversal**, we process all of a vertex's descendents before we move to an adjacent vertex. This concept is most easily seen when the graph is a tree. In Figure 15-48 we show the *tree* preorder traversal-processing sequence, one of the standard depth-first traversals.

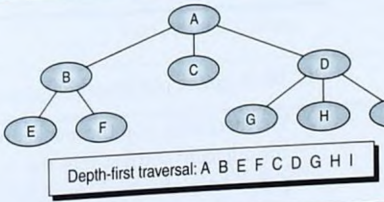


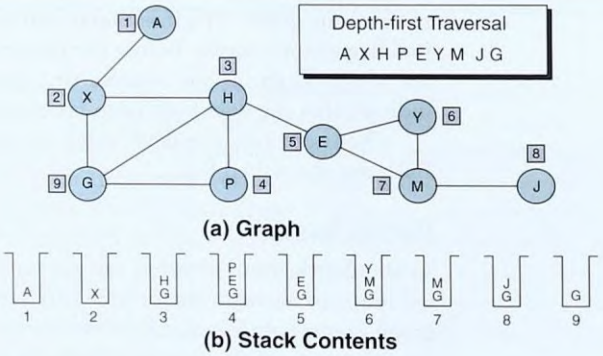
FIGURE 15-48 Depth-first Traversal of a Tree

In a similar manner, the depth-first traversal of a graph starts by processing the first vertex of the graph. After processing the first vertex, we select any vertex adjacent to the first vertex and process it. As we process each vertex, we select an adjacent vertex until we reach a vertex with no adjacent entries. This is similar to reaching a leaf in a tree. We then back out of the structure, processing adjacent vertices as we go. It should be obvious that this logic requires a stack (or recursion) to complete the traversal.

The order in which the adjacent vertices are processed depends on how the graph is physically stored. Because we are using a stack, however, the traversal processes adjacent vertices in descending, or last in–first out (LIFO), order.

In the depth-first traversal, all of a node's descendents are processed before moving to an adjacent node.

Let's trace a depth-first traversal through the graph in Figure 15-49. The number in the box next to a vertex indicates the processing order. The stacks below the graph show the stack contents as we work our way down the graph and then as we back out.



(a) Graph

(b) Stack Contents

FIGURE 15-49 Depth-first Traversal of a Graph

1. We begin by pushing the first vertex, A, into the stack.
2. We then loop, pop the stack, and, after processing the vertex, push all of the adjacent vertices into the stack. To process X at step 2, therefore, we pop X from the stack, process it, and then push G and H into the stack, giving the stack contents for step 3 as shown in Figure 15-49(b)—H G.
3. When the stack is empty, the traversal is complete.

Breadth-first Traversal

In the **breadth-first traversal** of a graph, we process all adjacent vertices of a vertex before going to the next level. We saw the breadth-first traversal earlier in the chapter. Looking at the tree in Figure 15-50, we see that its breadth-first traversal starts at level 0 and then processes all the vertices in level 1 before going on to process the vertices in level 2.

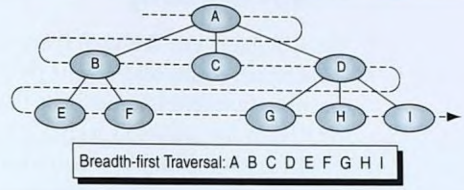


FIGURE 15-50 Breadth-first Traversal of a Tree

The breadth-first traversal of a graph follows the same concept. We begin by picking a starting vertex (A); after processing it we process all of its adjacent vertices (BCD). After we process all of the first vertex's adjacent vertices, we pick its first adjacent vertex (B) and process all of its vertices, then the second adjacent vertex (C) and all of its vertices, and so forth until we are finished.

The breadth-first traversal uses a queue rather than a stack. As we process each vertex, we place all of its adjacent vertices in the queue. Then, to select the next vertex to be processed, we delete a vertex from the queue and process it. Let's trace this logic through the graph in Figure 15-51.

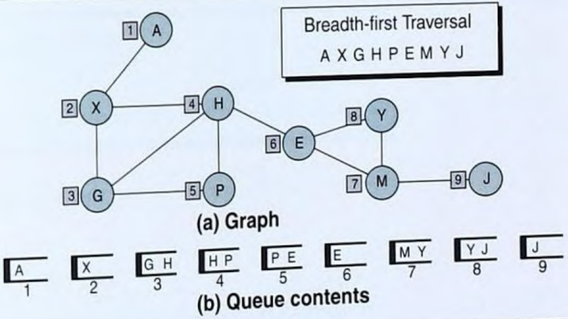


FIGURE 15-51 Breadth-first Traversal of a Graph

1. We begin by enqueuing vertex A in the queue.
2. We then loop, dequeuing the queue and processing the vertex from the front of the queue. After processing the vertex, we place all of its adjacent vertices into the queue. Thus, at step 2 in Figure 15-51(b), we dequeue vertex X, process it, and then place vertices G and H in the queue. We are then ready for step 3, in which we process vertex G.
3. When the queue is empty, the traversal is complete.

In the breadth-first traversal, all adjacent vertices are processed before processing the descendents of a vertex.

15.7 Software Engineering

Because linear lists are useful structures, programmers use them in many applications. Rather than rewrite their functions each time we need them, we can write functions once and put them in a library. Then when we need to use a linear list, we simply include the library. The name given to a complete set of functions built like this is **abstract data type** (ADT). To present the concept, we need to define a few new terms.

Atomic and Composite Data

Atomic data are data that we choose to consider as a single, nondecomposable, entity. For example, the integer 4562 may be considered as a single integer value. Of course, you can decompose it into digits, but the decomposed digits do not have the same characteristics of the original integer; they are four one-digit integers in the range 0 to 9.

An **atomic data type** is a set of atomic data having identical properties. These properties distinguish one atomic data type from another. Atomic data types are defined by a set of values and a set of operations that act on the values.

Atomic Data Type

1. A set of values.
2. A set of operations on the values.

For example, we can define the following atomic data types:

```
int
   VALUES:       -∞, …, -2, -1, 0, 1, 2, …, ∞
   OPERATIONS:    *, +, -, %, /, ++, --, <, >, …
float
   VALUES:       -∞, …, 0.0, …. ∞
   OPERATIONS:   *, +, -, /, <, >, …
char
   VALUES:       \0, …, 'A', 'B', …, 'a', 'b', …, \127
   OPERATIONS:   <, >, …
```

The opposite of atomic data is **composite data**. Composite data can be broken into subfields that have meaning. As an example of a composite data item, consider your telephone number. A telephone number actually has three different parts. First is the area code. Then, what you consider to be your phone number is actually two different data items, a prefix consisting of three digits and the number within the prefix, consisting of four digits. Years ago, these prefixes were names such as DAvenport and CYpress.

Data Structure

A **data structure** is a collection of elements and the relationships among them. Data structures can be nested. That is, we can have a data structure that consists of other data structures.

Data Structure

1. A combination of elements, each of which is either a data type or another data structure.
2. A set of associations or relationships (structure) involving the combined elements.

For example, we can define the two structures *array* and *struct,* as shown in Table 15-2.

array	struct
A homogeneous combination of data structures.	A heterogeneous combination of data structures.
Position association.	No association.

TABLE 15-2 Two Structures

Most programming languages support several data structures. In addition, modern programming languages allow programmers to create new data structures that are not available in the language they are using. In C, this is done with *struct* (see Chapter 12).

Abstract Data Type

Generally speaking, programmers' capabilities are determined by the tools in their tool kits. These tools are acquired by education and experience. Your knowledge of C is one of your tools. As you continue to study subjects such as data structures, file management, and systems analysis, your tools will increase. Abstract data types (ADT) are another tool to add to your tool kit.

The first computer programming languages had no ADTs. To read a file, programmers wrote the code to read the file device. It did not take long to realize that programmers were writing the same code over and over again. So they created what is known today as an ADT. A programmer wrote the code to read a file and placed it in a library for all programmers to use.

This concept is found in C today. For example, the standard input/output library is an ADT. It has data structures and a set of operations that can be used to read and write data.

With an ADT, the user is not concerned with *how* the task is done but rather with *what* it can do. In other words, the ADT consists of a set of prototype definitions that allow the programmer to use the functions while hiding the implementation. This generalization of operations with unspecified implementations is known as abstraction. We abstract the essence of the process and leave the implementation details hidden.

In the concept of abstraction

- ❑ We know *what* a data type can do.
- ❑ *How* it is done is hidden.

Consider the concept of a list. At least three data structures can support a list. We can use an array, a linear list, or a file. If we place the list in an abstract data type, the user should not be aware of the structure we use. As long as data can be inserted and retrieved, it should make no difference how we store the data. Figure 15-52 shows several structures that might be used to hold a list.

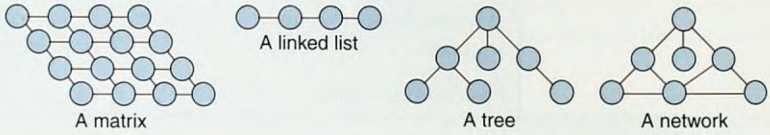


FIGURE 15-52 Structures for Holding a List

An **abstract data type** is formally defined as a data declaration packaged together with the operations that are allowed on the data type. In other words, we *encapsulate* the data and the operations on data, and we *hide* their implementation from the user.

Abstract Data Type

1. Declaration of data.
2. Declaration of operations.

The ADT definition implies two attributes for ADTs:

1. *The structures are opaque.* We can use them without knowing how they are implemented.
2. *The operations are opaque.* We know what they do; we don't know how they do it.

We cannot overstress the importance of hiding the implementation. For example, the programmer should not have to know the data structure to use the ADT. This is a common fault in many implementations that keep the ADT from being fully portable to other applications. Fortunately, C's rich library capability gives us the tools to fully implement any ADT.

A Model for an Abstract Data Type

The ADT model is shown in Figure 15-53. The model is represented by the blue area with an irregular outline. Inside are two different aspects of the model: the data structure and the operational functions. Both are entirely contained in the model and are not within the user's scope. However, the data structure is available to all the ADT's operations as needed, and an operation may call on other functions to accomplish its task. In other words, the data structure and the functions are within scope of each other.

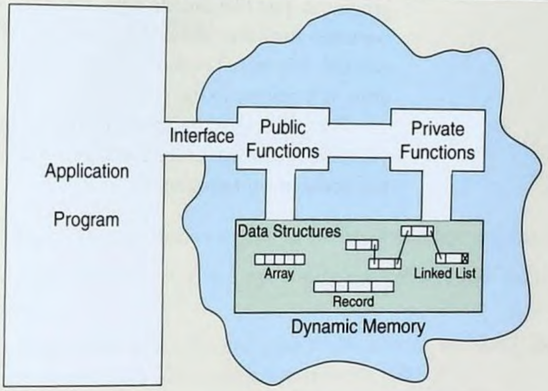


FIGURE 15-53 Abstract Data Type Model

Data flows in and out of the ADT through the operation headers represented by the rectangular "Interface" pathway. The interface is also the pathway for the ADT functions. For instance, a linear list ADT would have operations that we saw in this chapter, such as insert, remove, and search. In C, these operations are defined as prototype header declarations that are visible to the user. Note, however, that only the "Public" functions are available to the application program; the "Private" functions are totally contained with the ADT and can only be used by other functions within the ADT.

To use the ADT in our program, we need to create a user header file. When we create a header file for a linear list ADT, we give its header file a name, such as `linklist.h`. To include it in our program, we would use a preprocessor directive such as:

```
#include "linklist.h"
```

ADT Data Structure

When the list is controlled entirely by the program, it is often implemented using simple structures such as those shown in this chapter. Since the ADT must hide the implementation from the user, however, all data about the structure must be maintained inside the ADT. But just encapsulating the structure in the ADT is not sufficient. In addition, multiple versions of the structure must be able to coexist. This means that we must hide the implementation from the user while storing data about the structure in the user's program.

You have seen this concept before. When you create a file, you use the predefined structure `FILE`. Defining a file in your program creates a file structure that becomes a part of your program. We can do the same thing with the ADT. Each ADT must have a defined type that the users can define in their programs. Just like the file type, the ADT type is a pointer to a structure that contains attributes about the structure. When the ADT attribute structure is created, it is stored in the heap. The only structural element in the user's program is a pointer to the structure.

This short description of ADTs just begins to introduce the topic. When you take a data structure class, you may have the opportunity to create and use some in your programs.

15.8 Tips and Common Programming Errors

1. The link field in the last node of a linked list must have a null value.
2. Memory must be allocated for a node before you add the node to a linked list.
3. Be sure to free memory after you delete a node from a linked list.
4. You must create an empty linked list (by assigning *NULL* to the header pointer) before using the functions introduced in this chapter.
5. Remember that a null link means there is no pointer; therefore, you cannot use it to dereference another node (or any other object). For example, the following code creates a run-time error because when the loop terminates, the value of pCur is *NULL*.

```
while (pCur != NULL)
{
 ...
 pCur = pCur->link;
 printf("%d", pCur->data.member_name);        // ERROR
} // while
```

6. It is a logic error to allocate a node in the heap and not test for overflow.
7. It is a logic error to refer to a node after its memory has been released with *free*.
8. It is a logic error to set the only pointer to a node to *NULL* before the node has been freed. The node is irretrievably lost.
9. It is a logic error to delete a node from a linked list without verifying that the node contains the target of the delete.
10. It is a logic error to fail to set the head pointer to the new node when a node is added before the first node in a linked list. The new node is irretrievably lost.
11. It is a logic error to update the link field in the predecessor to a new node before pointing the new node to its logical successor. This error results in a never-ending loop next time the list is traversed.
12. It is a logic error to fail to set the link field in the last node to *NULL*. This causes the next traversal to run off the end of the list.
13. It is a potential logic error to use the node pointer in a linked list search before testing for a null pointer. In the following statement, the compares need to be reversed to prevent an invalid memory access.

```
while ((*pCur)->data.key < target
       && *pCur != NULL)
```

15.9 Key Terms

abstract data type (ADT)
ancestor
arc
atomic data
binary search tree (BST)
binary tree
breadth-first traversal
child
chronological list
composite data
connected
cycle
data structure
degree
depth
depth-first traversal
dequeue
descendant
direct graph
edge
enqueue
first in–first out (FIFO)
front
graph
indegree
inorder traversal
key-sequence list
last in–first out (LIFO)
line
linear list
link
linked list
list
loop
metadata
node
nonlinear list
outdegree
parent
pop
push
queue
rear
root
self-referential structure
stack
strongly connected
top
traversal
tree
underflow
vertex
weakly connected
weighted graph

15.10 Summary

- List can be divided into linear lists and non-linear lists.
- Linear lists can be divided into general lists, stacks, and queues.
- Non-linear lists can be divided into trees and graphs.
- In a general linear list, data can be inserted anywhere, and there are no restrictions on the operations that can be used to process the list.
- Four common operations are associated with general linear lists: insertion, deletion, retrieval, and traversal.
- When we want to insert into a general linear list, we must consider four cases:

  a. adding to the empty list
  b. adding at the beginning

c. adding at the middle

d. adding at the end

- When we want to delete a node from a general linear list, we must consider two cases: delete the first node or delete any other node.
- Traversing a general linear list means going through the list, item by item, and processing each item.
- A stack is a linear list in which all additions are restricted to one end, called the top. A stack is also called a LIFO list.
- We discuss two basic operations for stack: push and pop.
- Push stack adds an item to the top of the stack. After the push, the new item becomes the top.
- Pop stack removes the item at the top of the stack. After the top the next item, if any, becomes the top.
- A queue is a linear list in which data can be inserted at one end called the rear, and deleted from the other end, called the front.
- We discussed only two main operations for queues in this chapter: enqueue and dequeue.
- The enqueue operation inserts an element at the rear of the queue.
- The dequeue operation deletes the element at the front of the queue.
- A tree consists of a finite set of elements called nodes and a finite set of directed lines called branches that connect the nodes.
- A node in a tree can be a parent, a child, or both. Two or more nodes with the same parents are called siblings.
- A binary tree is a tree in which no node can have more than two children.
- In a preorder traversal of a tree, we process the left subtree first, followed by the root and then the right subtree.
- In the inorder traversal of a tree, we process the root first, followed by the left subtree and then followed by the right subtree.
- A binary search tree is a binary tree with the following properties:

  a. All items in the left subtree are less than the root.

  b. All items in the right subtree are greater than or equal to the root.

  c. Each subtree is itself a binary search tree
- A graph is a collection of nodes, called vertices, and a collection of line segments, called edges or arcs, connecting a pair of nodes.
- Graphs may be directed or undirected. In a directed graph, or digraph, each line has a direction. In an undirected graph, there is no direction on the lines. A line in a directed graph is called an arc.
- There are two standard graph traversals: depth first and breadth first.

15.11 Practice Set

Review Questions

1. In a linked list implementation, each node must contain data and a link field.

   a. True
   b. False

2. In a linked list implementation, there is a need for a head pointer to identify the beginning of the list.

   a. True
   b. False

3. In a linked list implementation, the first step in adding a node to a list is to allocate memory for the new node.

   a. True
   b. False

4. The C language provides a list structure.

   a. True
   b. False

5. In an array, the sequentiality of a list is maintained by ________________.

   a. the order structure of elements
   b. a pointer to the next element
   c. either a or b
   d. neither a nor b

6. ________________ is an ordered collection of data in which each element contains the location of the next element or elements.

   a. An array
   b. A structure
   c. A linked list
   d. None of the above

7. A stack is a ________________ structure.

   a. first in–last out
   b. last in–first out
   c. first in–first out
   d. last in–last out

8. A queue is a ________________ structure.

   a. first in–last out
   b. last in–first out
   c. first in–first out
   d. last in–last out

9. A general linear list is a list in which operations, such as retrievals, insertions, changes, and deletions, can be done ______________.

    a. anywhere in the list
    b. only at the beginning
    c. only at the end
    d. only at the middle

10. A stack is a list in which operations, such as retrievals, insertions, changes, and deletions, can be done ______________.

    a. anywhere in the list
    b. only at the top
    c. only at the base (bottom)
    d. only at the middle

11. A queue is a list in which operations, such as retrievals, insertions, changes, and deletions, can be done ______________.

    a. only at the beginning
    b. only at the end
    c. only at the middle
    d. none of the above

12. A ______________ is a last in–first out (LIFO) data structure in which insertion and deletions are restricted to one end, called the top.

    a. stack
    b. queue
    c. tree
    d. binary tree

13. A ______________ is a first in–first out (FIFO) data structure in which insertions are restricted to one end, called the rear, and deletions are restricted to another end, called the front.

    a. stack
    b. queue
    c. tree
    d. binary tree

14. To add an element to a stack, we use the ______________ operation.

    a. pop
    b. push
    c. enqueue
    d. dequeue

15. To delete an element from a stack, we use the ______________ operation.

    a. pop
    b. push
    c. enqueue
    d. dequeue

16. To add an element to a queue, we use the ________________ operation.

a. pop
b. push
c. enqueue
d. dequeue

17. To delete an element from a queue, we use the ________________ operation.

a. pop
b. push
c. enqueue
d. dequeue

18. Data that consist of a single, nondecomposable entity are known as ________________.

a. atomic data
b. composite data
c. derived data
d. standard data
e. structured data

19. A(n) ________________ is a collection of elements and the relationship among them.

a. abstract data type
b. array
c. data structure
d. standard type
e. type definition

Exercises

20. Imagine we have the general list shown in Figure 15-54. Show what happens if we apply the following statement to this general list:

```
pList = pList -> link;
```

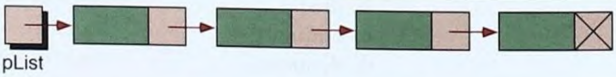


FIGURE 15-54 Figure for Exercise 20

What is the problem with using this kind of statement? Does it justify the need for two walking pointers (`pPre` and `pCur`) that we introduced in the text?

21. Imagine we have the general list shown in Figure 15-55. As discussed in "Search Linear List," the search function needs to be able to pass back both the location of the predecessor (`pPre`) and the location of the current (`pCur`) node based on search criteria. A typical search design is shown in Figure 15-55.

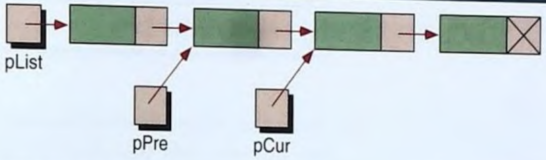


FIGURE 15-55 Figure for Exercise 21

The following code to set `pPre` and `pCur` contains a common error. What is it, and how should it be corrected? (*Hint*: What are the contents of these pointers at the beginning of the search?)

```
pCur = pCur->link;
pPre = pPre->link;
```

22. Imagine we have a dummy node at the beginning of a general list. The dummy node does not carry any data. It is not the first data node; it is an empty node. Figure 15-56 shows a general list with a dummy node. Write the code to delete the first node (the node after the dummy node) in the general list.

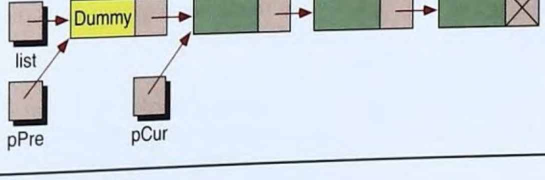


FIGURE 15-56 Figure for Exercise 22

23. Write the code to delete a node in the middle of a general list with the dummy node (see Exercise 22). Compare your answer with the answer to Exercise 22. Are they the same? What do you conclude? Does the dummy node simplify the operation on a general list? How?

24. Figure 15-57 shows an empty general list with a dummy node. Write the code to add a node to this empty general list.

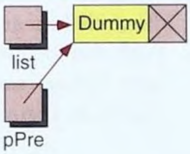


FIGURE 15-57 Figure for Exercise 24

25. Write the statements to add a node in the middle of a general list with the dummy node (see Exercise 22). Compare your answer with the answer to Exercise 24. Are they the same? What do you conclude? Does the dummy node simplify the operation on a general list? How?

26. Imagine we have the two general lists shown in Figure 15-58. What would happen if we apply the following statement to these two lists?

```
list1 = list2;
```

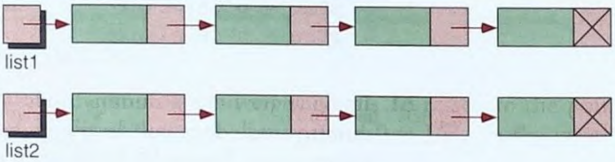


FIGURE 15-58 Figure for Exercise 26

27. What would happen if we apply the following statements to the two lists in Exercise 26?

```
temp = list1;
while (temp->link != NULL)
   temp = temp->link;
temp->link = list2;
```

28. Imagine we have the general list shown in Figure 15-59. What would happen if we apply the following statements to this list?

```
temp = list;
while (temp->link != NULL)
   temp = temp->link;
temp->link = list;
```

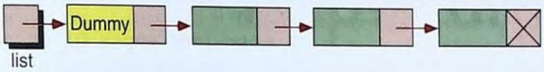


FIGURE 15-59 Figure for Exercise 28

29. Imagine we have two empty stacks of integers, s1 and s2. Draw a picture of each stack after the following operations:

```
Algorithm Exercise 29
pushStack (s1, 3)
pushStack (s1, 5)
pushStack (s1, 7)
pushStack (s1, 9)
pushStack (s1, 11)
pushStack (s1, 13)
loop not emptyStack (s1)
   popStack (s1, x)
   pushStack (s2, x)
end loop
End Exercise 33
```

30. Imagine you have a stack of integers, S, and a queue of integers, Q. Draw a picture of S and Q after the following operations:

```
Algorithm Exercise 30
pushStack (S, 3)
pushStack (S, 12)
enqueue (Q, 5)
enqueue (Q, 8)
popStack (S, x)
pushStack (S, 2)
enqueue (Q, x)
dequeue (Q, y)
pushStack (S, x)
pushStack (S, y)
End Exercise 30
```

31. Imagine that the contents of queue Q1 and queue Q2 are as shown. What would be the contents of Q3 after the following code is executed? The queue contents are shown front (left) to rear (right).

```
Algorithm Exercise 31
Q3 = createQueue
count = 1
```

continued

```
 3 y = 0
 4 loop (not empty Q1 or not empty Q2)
    1  dequeue (Q1, x)
    1  if (count > y)
       1  dequeue (Q2, y)
    2  end if
    3  if (y equal count)
       1  enqueue (Q3, x)
    4  end if
    5  count = count + 1
 5 end loop
End Exercise 31
```

Q1: 42 30 41 31 19 20 25 14 10 11 12 15

Q2: 4 5 6 10 13

32. Draw all possible non-similar binary trees with three nodes (A, B, C).

33. Draw all possible binary search trees for the data elements 5, 9, and 12.

34. Create a binary search tree using the following data entered as a sequential set:

```
14, 23, 7, 10, 33, 56, 80, 66, 70
```

35. Create a binary search tree using the following data entered as a sequential set:

```
7, 10, 14, 23, 33, 56, 66, 70, 80
```

36. Insert 44 and 50 into the tree created in Exercise 34.

37. Insert 44 and 50 into the tree created in Exercise 35.

Problems

38. Write a program that reads a list of integers from the keyboard, creates a general list from them, and prints the result.

39. Write a function that accepts a general list, traverses it, and returns the key of the node with the minimum key value.

40. Write a function that traverses a general list and deletes all nodes whose keys are negative.

41. Write a function that traverses a general list and deletes all nodes that are after a node with a negative key.

42. Write a function that traverses a general list and deletes all nodes that are before a node with a negative key.

43. Rewrite the function `deleteNode` (see Program 15-2) using a general list with a dummy node.

44. Rewrite the function `searchList` (see Program 15-3) using a general list with a dummy node.

45. Write a function that returns a pointer to the last node in a general list.

46. Write a function that appends two general lists together.

47. Write a function that appends a general list to itself.

48. One of the applications of a stack is to backtrack—that is, to retrace its steps. As an example, imagine we want to read a list of items, and each time we read a negative number we must backtrack and print the five numbers that come before the negative number and then discard the negative number. Use a stack to solve this problem. Read the numbers and push them into the stack (without printing them) until a negative number is read. At this time, stop reading and pop five items from the stack and print them. If there are fewer than five items in the stack, print an error message and stop the program. After printing the five items, resume reading data and placing them in the stack. When the end of the file is detected, print a message and the items remaining in the stack. Test your program with the following data:

```
1 2 3 4 5 -1 1 2 3 4 5 6 7 8 9 10 -2 11 12 -3 1 2 3 4 5
```

49. Write a function called `copyStack` that copies the contents of one stack into another. The algorithm passes two stacks, the source stack and the destination stack. The order of the stacks must be identical. (*Hint*: Use a temporary stack to preserve the order.)

50. Write a function, `catStack`, that concatenates the contents of one stack on top of another.

51. Write a function to check whether the contents of two stacks are identical. Neither stack should be changed. You need to write a function that prints the contents of a stack to verify that your function works.

52. Write a function called `copyQueue` that copies the contents of one queue to another.

53. Write a function called `stackToQueue` that creates a queue from a stack. After the queue has been created, the top of the stack should be the front of the queue and the base of the stack should be the rear of the queue. At the end of the function, the stack should be empty.

54. Given a queue of integers, write a function that calculates and prints the sum and the average of the integers in the queue without changing the contents of the queue.

55. Given a queue of integers, write a function that deletes all negative integers without changing the order of the remaining elements in the queue.

56. Write a function that calculates and passes up to the calling function the sum and average of the nodes in a tree.

57. Write a function that counts the number of leaves in a binary tree.

58. Write a function to find the smallest node in a binary search tree.

Projects

59. Write a program that reads a file and builds a key-sequenced general list. After the list is built, display it on the monitor. You may use any appropriate data structure, but it must have a key field and data. Two possibilities are a list of your favorite CDs or your friends' telephone numbers.

60. Write a program to read a list of students from a file and create a general list. Each entry in the general list is to have the student's name, a pointer to the next student, and a pointer to a general list of scores. You may have up to four scores for each student.The program initializes the student list by reading the students' names from the text file and creating null scores lists. After building the student list, it loops through the list, prompting the user to enter the scores for each student. The scores prompt is to include the name of the student.

61. Write a stack and queue test driver. A test driver is a program created to test functions that are to be placed in a library. Its primary purpose is to completely test functions; therefore, it has no application use. The functions to be tested are push stack, pop stack, enqueue, and dequeue. You may include other stack and queue functions as required. All data should be integers. You need two stacks and two queues in the program, as described below.

a. Input stack: used to store all user input
b. Input queue: used to store all user input
c. Output stack: used to store data deleted from input queue
d. Output queue: used to store data deleted from input stack

Use a menu-driven user interface that prompts the user to select either insert or delete. If an insert is requested, the system should prompt the user for the integer to be inserted. The data are then inserted into the input stack and input queue. If a delete is requested, the data are deleted from both structures: the data popped from the input stack are enqueued in the output queue, and the data dequeued from the input queue are pushed into the output stack.

Processing continues until the input structures are empty. At this point print the contents of the output stack while deleting all of its data. Label this output "Output Stack," then print all of the data in the output queue while deleting all of its data. Label this output "Output Queue." Your output should be formatted as shown below.

```
Output Stack: 18  9 13  7  5  1
Output Queue:  7 13  9 18  5  1
```

Test your program with the following operations:

```
1 input 1   5 delete     9 input 6   13 input 8
2 input 2   6 input 0   10 delete    14 delete
3 delete    7 input 5   11 input 7   15 delete
4 input 3   8 delete    12 delete    16 delete
```

In addition to the computer output from your test, write a short report (less than one page) describing what structural concepts were demonstrated by your output.

Appendix A
Character Sets

Computers use numbers. They store characters by assigning a number to each one. The C language was originally designed using the American Standard Code for Information Interchange (ASCII), which was the standard of the time. Based on the Latin alphabet, ASCII uses 128 characters (valued from 0 to 127) stored as 7-bit numbers. This is enough to handle the lower and uppercase letters, digits, most common punctuation characters, and some control characters. Later, an attempt was made to extend the ASCII character set to 8 bits. The new code, which was called **Extended ASCII**, was never internationally standardized.

To overcome the limitations inherent in ASCII and Extended ASCII, the International Standard Organization (ISO) and the Unicode Consortium (a consortium of manufacturers of multilingual software) created a universal encoding system to provide a comprehensive character set. The Unicode consortium created Unicode. ISO created the **Universal Character Set (UCS),** designated ISO 10646 standard.

The main UCS set, sometimes referred to as UCS-4, is a four-byte code. A subset of UCS-4, called UCS-2, is a two-byte code that is compatible with UCS-4; it contains the first 32,364 values in UCS-4.

Unicode was originally a two-byte character set. The early versions of Unicode were only compatible with UCS-2. Unicode version 3 is a four-byte code and is fully compatible with UCS-4. The compatibility of Unicode with UCS-4 makes Unicode compatible with ASCII and Extended ASCII.

The ASCII set, which is now called ***Basic Latin,*** is included in UCS-4 and Unicode with the upper 25 bits set to zero. The Extended ASCII, which is now called Latin-1, is also included with 24 upper bits set to zero. Figure A-1 shows the compatibility between the different systems.

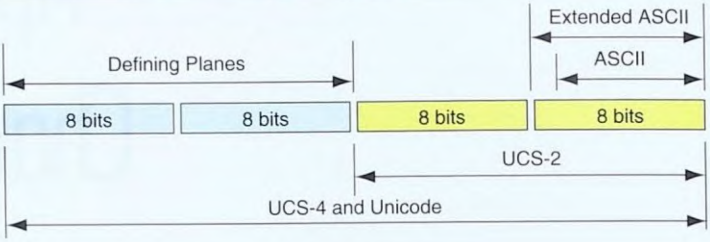


FIGURE A-1 USC and Unicode Compatibility

A.1 Unicode

The prevalent code today is **Unicode**. Each character or symbol in this code is defined by a 32-bit number. The code can define up to 2^{32} (4,294,967,296) characters or symbols. The presentation uses hexadecimal digits in the following format:

U-XXXXXXXX

where each X is a hexadecimal digit. Therefore, the numbering goes from U-00000000 to U-FFFFFFFF.

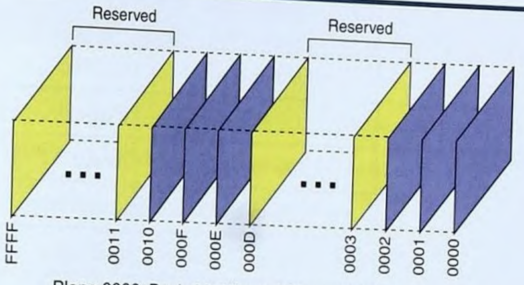
Planes

Unicode divides the whole space code into planes. The most significant 16 bits define the plane, which means we can have 65,536 planes. For plane 0, the most significant 16 bits are 0s (0x0000); in plane 1, the bits are 0x0001; in plane 2, they are 0x0002; and so on until in the plane 65,536, they are 0xFFFF.

Each plane can define up to 65,536 character or symbols. Figure A-2 shows the structure of Unicode spaces and its planes.

Basic Multilingual Plane (BMP)

Basic Multilingual Plane, Plane 0, is designed to be compatible with the previous 16-bit Unicode version and UCS-2. The most significant 16 bits in this plane are all zeros. The codes are normally shown as U+XXXX with the understanding that XXXX defines only the least significant 16 bits. This plane mostly defines character sets in different languages with the exception of some codes used for control or other special characters. Table A-1 shows the main classification of codes in Plane 0.



Plane 0000: Basic Multilingual Plane (BMP)
Plane 0001: Supplementary Multilingual Plane (SMP)
Plane 0002: Supplementary Ideographic Plane (SIP)
Plane 000E: Supplementary Special Plane (SSP)
Plane 000F: Private Use Plane (PUP)
Plane 0010: Private Use Plane (PUP)

FIGURE A-2 Unicode Planes

Range	Description
A-Zone (Alphabetical Characters and Symbols)	
U+0000 to U+00FF	Basic Latin and Latin-1
U+0100 to U+01FF	Latin Extended
U+0200 to U+02FF	IPA Extension, and Space Modifier Letters
U+0300 to U+03FF	Combining Diacritical Marks, Greek
U+0400 to U+04FF	Cyrillic
U+0500 to U+05FF	Armenian, Hebrew
U+0600 to U+06FF	Arabic
U+0700 to U+08FF	Reserved
U+0900 to U+09FF	Devanagari, Bengali
U+0A00 to U+0AFF	Gurmukhi, Gujarati
U+0B00 to U+0BFF	Oriya, Tamil
U+0C00 to U+0CFF	Telugu, Kannda
U+0D00 to U+0DFF	Malayalam

continued

TABLE A-1 Unicode BMP

Range	Description
U+0E00 to U+0EFF	Thai, Lao
U+0F00 to U+0FFF	Reserved
U+1000 to U+10FF	Georgian
U+1100 to U+11FF	Hangul Jamo
U+1200 to U+1DFF	Reserved
U+1E00 to U+1EFF	Latin extended additional
U+1F00 to U+1FFF	Greek extended
U+2000 to U+20FF	Punctuation, sub/superscripts, currency, marks
U+2100 to U+21FF	Letter-like symbols, number forms, arrows
U+2200 to U+22FF	Mathematical operations
U+2300 to U+23FF	Miscellaneous technical symbols
U+2400 to U+24FF	Control pictures, OCR, and enclosed alphanumeric
U+2500 to U+25FF	Box drawing, block drawing, and geometric shapes
U+2600 to U+26FF	Miscellaneous symbols
U+2700 to U+27FF	Dingbats and Braille patterns
U+2800 to U+2FFF	Reserved
U+3000 to U+30FF	CJK symbols and punctuation, Hiragana, Katakana
U+3100 to U+31FF	Bopomofo, Hangul Jamo, CJK Miscellaneous
U+3200 to U+32FF	Enclosed CJK letters and months
U+3300 to U+33FF	CJK compatibility
U+3400 to U+4DFF	Hangul
I-Zone (Ideographic Characters)	
U+4E00 to U+9FFF	CJK Unified Ideographic
O-Zone (Open)	
U+A000 to U+DFFF	Reserved
R-Zone (Restricted Use)	
U+E000 to U+F8FF	Private Use

continued

TABLE A-1 Unicode BMP *(continued)*

Range	Description
U+F900 to U+FAFF	CJK Compatibility Ideographs
U+FB00 to U+FBFF	Arabic Presentation Form-A
U+FC00 to U+FDFF	Arabic Presentation Form-B
U+FE00 to U+FEFF	Half marks, small forms
U+FF00 to U+FFFF	Half-width and full-width forms

TABLE A-1 Unicode BMP *(continued)*

Supplementary Multilingual Plane (SMP)

Supplementary Multilingual Plane, Plane 1, is designed to provide more code for those multilingual characters that are not included in the BMP plane.

Supplementary Ideographic Plane (SIP)

Supplementary Ideographic Plane, Plane 2, is designed to provide code for ideographic symbols, any symbol that primarily denotes an idea (or meaning) in contrast to a sound (or pronunciation).

Supplementary Special Plane (SSP)

Supplementary Special Plane, Plane 14 (0x000E), is used for special characters not found in the Basic Latin or Basic Latin-1 codes.

Private Use Planes (PUP)

Private Use Planes, Planes 15 (0x000F) and 16 (0x0010), are reserved for private use.

A.2 ASCII

The American Standard Code for Information Interchange (ASCII) is a seven-bit code, that was designed to provide code for 128 symbols. Today, ASCII or Basic Latin, is part of the Unicode. It occupies the first 128 codes in Unicode (U-00000000 to U-0000007F). Table A-2 contains the decimal, hexadecimal, and graphic codes (symbols) with an English interpretation, if appropriate. The codes in hexadecimal only define the two least significant digits in Unicode. To find the actual code, we prepend 000000 in hexadecimal to the code. The decimal value shows the integer value of each symbol when converted to an integer.

Decimal	Hex	Symbol	Interpretation
0	00	null	Null value
1	01	SOH	Start of heading
2	02	STX	Start of text
3	03	ETX	End of text
4	04	EOT	End of transmission
5	05	ENQ	Enquiry
6	06	ACK	Acknowledgment
7	07	BEL	Ring bell
8	08	BS	Backspace
9	09	HT	Horizontal tab
10	0A	LF	Line feed
11	0B	VT	Vertical tab
12	0C	FF	Form feed
13	0D	CR	Carriage return
14	0E	SO	Shift out
15	0F	SI	Shift in
16	10	DLE	Data link escape
17	11	DC1	Device control 1
18	12	DC2	Device control 2
19	13	DC3	Device control 3
20	14	DC4	Device control 4
21	15	NAK	Negative acknowledgment
22	16	SYN	Synchronous idle
23	17	ETB	End of transmission block
24	18	CAN	Cancel
25	19	EM	End of medium
26	1A	SUB	Substitute

continued

TABLE A-2 ASCII Codes

Decimal	Hex	Symbol	Interpretation
27	1B	ESC	Escape
28	1C	FS	File separator
29	1D	GS	Group separator
30	1E	RS	Record separator
31	1F	US	Unit separator
32	20	SP	Space
33	21	!	
34	22	"	Double quote
35	23	#	
36	24	$	
37	25	%	
38	26	&	
39	27	'	Apostrophe
40	28	(	
41	29	)	
42	2A	*	
43	2B	+	
44	2C	,	Comma
45	2D	–	Minus
46	2E	.	
47	2F	/	
48	30	0	
49	31	1	
50	32	2	
51	33	3	
52	34	4	
53	35	5	

continued

TABLE A-2 ASCII Codes *(continued)*

Decimal	Hex	Symbol	Interpretation
54	36	6	
55	37	7	
56	38	8	
57	39	9	
58	3A	:	Colon
59	3B	;	Semicolon
60	3C	<	
61	3D	=	
62	3E	>	
63	3F	?	
64	40	@	
65	41	A	
66	42	B	
67	43	C	
68	44	D	
69	45	E	
70	46	F	
71	47	G	
72	48	H	
73	49	I	
74	4A	J	
75	4B	K	
76	4C	L	
77	4D	M	
78	4E	N	
79	4F	O	
80	50	P	

continued

TABLE A-2 ASCII Codes *(continued)*

Decimal	Hex	Symbol	Interpretation
81	51	Q	
82	52	R	
83	53	S	
84	54	T	
85	55	U	
86	56	V	
87	57	W	
88	58	X	
89	59	Y	
90	5A	Z	
91	5B	[	Open bracket
92	5C	\	Backslash
93	5D	]	Close bracket
94	5E	^	Caret
95	5F	_	Underscore
96	60	`	Grave accent
97	61	a	
98	62	b	
99	63	c	
100	64	d	
101	65	e	
102	66	f	
103	67	g	
104	68	h	
105	69	i	
106	6A	j	
107	6B	k	

continued

TABLE A-2 ASCII Codes *(continued)*

Decimal	Hex	Symbol	Interpretation
108	6C	l	
109	6D	m	
110	6E	n	
111	6F	o	
112	70	p	
113	71	q	
114	72	r	
115	73	s	
116	74	t	
117	75	u	
118	76	v	
119	77	w	
120	78	x	
121	79	y	
122	7A	z	
123	7B	{	Open brace
124	7C	|	Bar
125	7D	}	Close brace
126	7E	~	Tilde
127	7F	DEL	Delete

TABLE A-2 ASCII Codes *(continued)*

Some Properties of ASCII

ASCII has some interesting properties that we need to briefly mention here:

1. The first code (0), is the null character, which represents the absence of any character.
2. The first 32 codes, 0 to 31, and the last code, 255, are control characters. They can be checked with the *iscontrol* function.
3. The space character, which is a printable character, is at position 32.

4. The uppercase letters start from 65 (A). The lowercase letters start from 97 (a). When numerically compared, uppercase letters are smaller than lowercase ones. This means that when we sort a list based on ASCII values, the uppercase letters show before the lowercase letters.
5. The uppercase and lowercase letters differ by only one bit in the seven-bit code. For example, character A is 1000001 (0x41) and character a is 1100001(0x61). The difference is bit 6, which is 0 in uppercase letters and 1 in lowercase letters. If we know the code for one case, we can find easily the code for the other by adding or subtracting 32 in decimal, 0x20 in hexadecimal, or flipping the sixth bit.
6. The uppercase letters are not immediately followed by lowercase letters. There are some punctuation characters in between.
7. Digits (0 to 9) begin at 48 (0x30). This means that if we want to change a numeric character to its face value as an integer, we need to subtract 48 from it.

A.3 Universal Encoding in C

The C language allows the use of the UCS-2 and UCS-4 codes in character constants, string constants, and identifiers. C uses two formats, one with four hexadecimal digits for UCS-2 and another with 8 hexadecimal digits for UCS-4 as shown below. Note that the two formats are compatible; \U0000*nnnn* is the same as \u*nnnn*.

```
\unnnn                                        // For UCS-2
\Unnnnnnnn                                    // For UCS-4
```

Program A-1 shows how we can print a graphical character using the universal encoding in C.

PROGRAM A-1 Example of Universal Encoding

```
/* Show the use of universal encoding
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Declarations
   char ch1 = '\u00A2';
   char ch2 = '\U000000A3';
```

continued

PROGRAM A-1 Example of Universal Encoding *(continued)*

```

// Statements
   printf ("Character ch1: %c\n", ch1);
   printf ("Character ch2: %c\n", ch2);

   return 0;
}  // main
```

```
Result:
Character ch1: ¢
Character ch2: £
```

Appendix B Keywords

The C language contains 37 keywords, also known as reserved words, that cannot be used as identifiers for functions, variables, or named constants. They are shown in Table B-1.

auto
_Bool
break
case
char
_Complex
const
continue
default
do
double
else
enum
extern
float
for
goto
if
_Imaginary
inline
int
long
register
restrict
return
short
signed
sizeof
static
struct
switch
typedef
union
unsigned
void
volatile
while

TABLE B-1 C Keywords

In addition to these keywords, we recommend that the macros and C++ keywords in Table B-2 also be treated as keywords.

and	compl	or
and_eq	complex	OR_EQ
bitand	imaginary	struct
bitor	not	xor_eq
bool	not_eq	

TABLE B-2 C Macros and C++ Keywords

Appendix C Flowcharting

A flowchart is a tool to show the logic flow of a program. Although it is generally considered a computer programming tool, it has been used for many other purposes. Programmers use flowcharts to design a complete program or just part of a program. Depending on the programming language, the parts can be called such things as procedures (Pascal), functions (C), or paragraphs (COBOL). We will use the general term *algorithm* to indicate any part of a program that needs design.

The primary purpose of a flowchart is to show the design of an algorithm. The flowchart also frees the programmer from the syntax and details of a programming language while allowing him or her to concentrate on the details of the problem to be solved.

A flowchart gives a pictorial representation of an algorithm—in contrast to another programming design tool, pseudocode, which provides a textual design solution. Both tools have their advantages, but a flowchart has the pictorial power that other tools lack. A new student of computer science must learn how to think about an algorithm before writing it. A flowchart is a tool for this pictorial thinking.

C.1 Auxiliary Symbols

A flowchart is a combination of symbols. Some symbols enhance the readability or functionality of the flowchart; they do not directly show instructions or commands. They show the start and stop points, the order and sequence of actions, and how one part of a flowchart is connected to another. These **auxiliary symbols** are shown in Figure C-1.

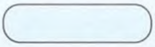
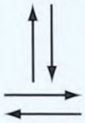
SYMBOL	NAME	APPLICATION
	Terminal	Shows the beginning or ending of an algorithm
	Flow Line	Shows the action order in an algorithm
	Connector	Shows connecting points in an algorithm

FIGURE C-1 Auxiliary Symbols

An oval shows the beginning or ending of an algorithm. When it is used to show the beginning of an algorithm, we write the word START inside the oval. When it is used to show the ending of an algorithm, we write the word STOP or RETURN in the oval.

One of the first rules of structured programming is that each algorithm should have only one entry point and one exit point. Thus, a good structured flowchart should have one and only one START and one and only one STOP. Ovals should be aligned to show clearly the flow of the actions in an algorithm. For example, a flowchart for a program that does nothing is shown in Figure C-2. This program starts and stops without doing anything.

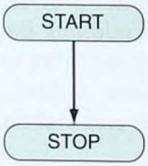


FIGURE C-2 Use of the Start-Stop Symbols

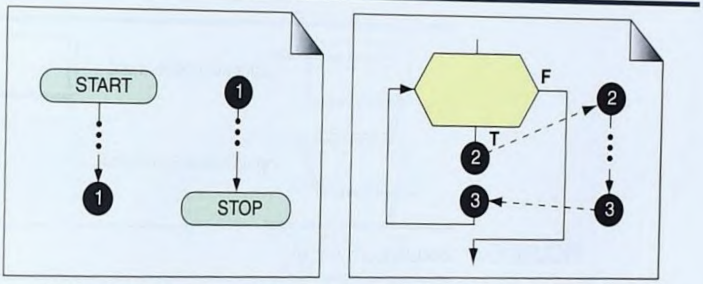
Flow Lines

Flow lines show the order or sequence of actions in a program. They connect symbols. Usually a symbol has some entering and some exiting lines. The START oval has only one exiting line. The STOP oval has only one entering line. We have already shown the use of flow lines in Figure C-2. We will show other flows in the examples that follow.

Connectors

We use only one **connector** symbol, a circle with a number in it, to show connectivity. It is used in two situations.

1. When we reach the end of a column or page, but our chart is not finished. In this case, at the bottom of the flow we use a connector to show that the flow continues at the top of the next column or page. This condition is shown in Figure C-3(a).



(a) Simple Flow Connectors (b) Insert Flow Connectors

FIGURE C-3 Use of Connectors

2. When we need to show logic details that do not fit in the flow. This is much like an inset in a map used to magnify a part of the map. In a flowchart, it magnifies part of the logic flow. In this case, we use two sets of connectors. The first set connects the main flow to the entry point in the magnified flow; the second set connects the end of the magnified flow to the return point in the main flow. This condition is shown in Figure C-3(b).

In either case, the number in the connector can be a simple serial number for an on-page flow, or it can be a combination of a page and number in the form page.number for an off-page flow.

C.2 Primary Symbols

Programmers use the primary symbols to show the instructions or actions needed to solve the problem presented in the algorithm. With these symbols, it is possible to represent all three structured programming constructs: sequence, decision, and repetition.

Sequence

Sequence statements simply represent a series of actions that must continue in a linear order. Although the actions represented in the sequence symbol may be very complex, such as an input or output operation, the logic flow must enter the symbol at the top and flow out at the bottom. Sequence symbols do not allow any decisions or flow changes within the symbol.

Flowcharts use four sequence symbols: assignment, input/output, module call, and compound statement. They are shown in Figure C-4.

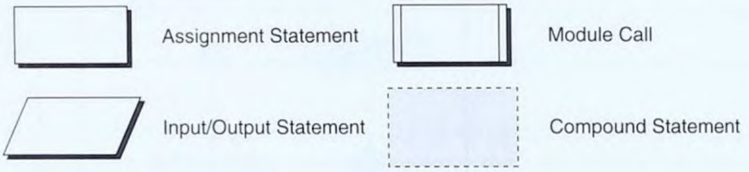


FIGURE C-4 Sequence Symbols

Null Statement

It is worth noting that *do nothing* is a valid statement. It is commonly referred to as a null statement. The null statement is considered a sequence statement, since it cannot change the flow direction of a program. There is no symbol for a null statement. It is simply a flow line. Figure C-2 is an example of a null statement.

Assignment Statement

The assignment statement is shown using a rectangle. Within the assignment symbol, the assignment operator is shown as a left-pointing arrow. At the right side of the arrow is an expression whose value must be stored in the variable at the left side. Figure C-5 shows an assignment statement.

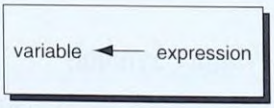


FIGURE C-5 The Assignment Statement

Input/Output Statement

A parallelogram shows any input or output, such as reading from a keyboard or writing on the system console. For example, an algorithm that reads the value of two variables from the keyboard and then writes their values on the screen in reverse order is shown in Figure C-6.

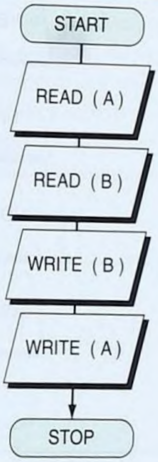


FIGURE C-6 Read and Write Statements

Module-Call Statement

The symbol used for a **module call** is a rectangle with two vertical bars inside. The flowchart for the called module must be somewhere else. In other words, each time you see a module call statement, look for another flowchart with the module name.

In C programs, the module-call symbol is used primarily for *void* functions. Functions that return values are shown as assignments or other expressions in a sequence symbol.

To show how a module call is used in a program, let's design the flowchart for a program that calculates and prints the average of three numbers. This example has two noteworthy points: First, the flowchart for the called module (AVRG) does not begin with START. Rather, it shows the name of the module and the parameter list. Also, the exit oval contains RETURN, indicating that it is not the end of the program but, rather, a return from a called module. The resulting flowchart is shown in Figure C-7.

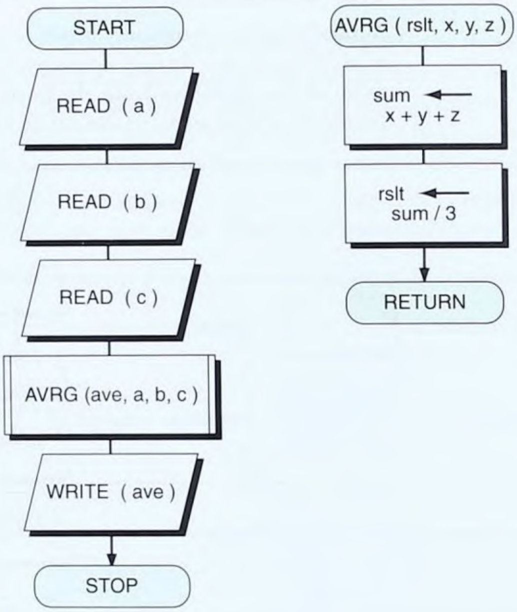


FIGURE C-7 Module Call Example

Compound Statement

Although flowcharts have no actual symbol to show a compound statement, we encapsulate all statements that make a compound statement in a broken-line rectangle. In a C program, compound statements represent a block of code, code that is enclosed in braces. In a flowchart, we use a dashed rectangle to enclose a compound statement. An example of this is shown in Figure C-9.

Selection Statements

Unlike the sequence statements, conditional statements can cause the flow of the program to change. They allow the execution of selected statements and the skipping of other statements. Structured programming has two selection statements: two-way and multiway selection.

Two-Way Selection

The two-way symbol is the diamond. When it is used to represent an *if...else* statement, the true condition logic is shown as the right leg of the logic flow, and the false condition, if present, is shown on the left leg of the logic flow. With the *if...else*, there must always be two logic flows, although often one of them is null. (Remember that the null statement is represented by a flow line; there is no symbol for null.) Finally, the statement ends with a connector where the true and false flows join. In this case, the connector has nothing in it.

Although you will often see decisions drawn with the flow from the bottom of the diamond, this is not good style. Even when one of the flows is null, it still must flow from the left or right sides of the diamond.

Figure C-8 shows the use of the decision symbol in the *if...else* statement. As we pointed out, a decision always has two branches. On each branch, we are allowed to have one and only one statement. Of course, the statement in each branch can be a null or a compound statement. But only one statement is allowed in each branch; not less, not more. Also remember that the whole figure is only one statement, not two or three; it is one *if...else* statement.

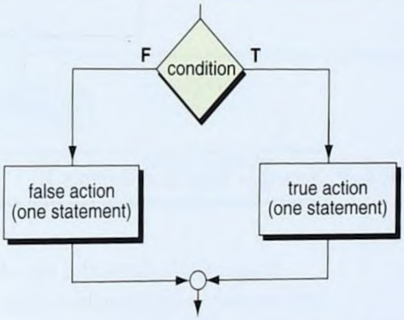


FIGURE C-8 *if ... else* Statement

Let's design an algorithm that reads an integer. If the integer's value is greater than 10, it subtracts 10 and writes the original number and the result. If the value is less than 10, it does nothing. The flowchart for this program is seen in Figure C-9.

Multiway Selection

The second application of the selection symbol used with structured programming is multiway selection. C's implementation of multiway selection is the *switch* statement. Actually, the multiway selection statement is nothing more than a shorthand notation for the *if...else* statement. If a language does not have a *switch* statement, the same logic is implemented using nested *if...else* statements or the *else-if* construct.

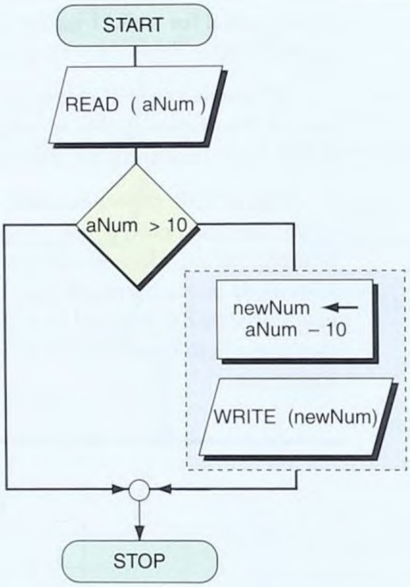


FIGURE C-9 Example: Read and Subtract 10

Figure C-10 shows the use of the *switch* statement. As you can see, we can have as many branches as we need. On each branch, we are allowed to have one, and only one, statement. Of course, the statement in each branch can be a null or a compound statement. But remember that only one statement in each branch is allowed; not less, not more. Also remember that the whole figure is only one statement, not two or three; it is one *switch* statement. (If these rules sound familiar, remember that the *switch* statement is nothing but a shorthand form of *if…else,* so the rules are the same for both.

Let's design an algorithm for a program that reads one character representing a letter grade and prints the corresponding grade point average (GPA). Figure C-11 shows this design.

Looping Statements

C has three looping statements: *for, while,* and *do…while*.

for Statement

The *for* statement is a counter-controlled loop. It is actually a complex statement that has three parts, any of which can be null: (1) the loop initialization, which normally sets the loop counter; (2) the limit test; and (3) the end-of-loop action statements, which usually increment a counter. Since the *for* statement is a pretest loop, the loop might not be executed. If the terminating condition is true at the start, the body of the *for* statement is skipped.

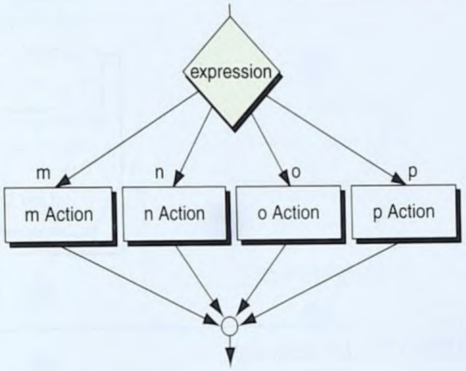


FIGURE C-10 Multiway Selection Statement

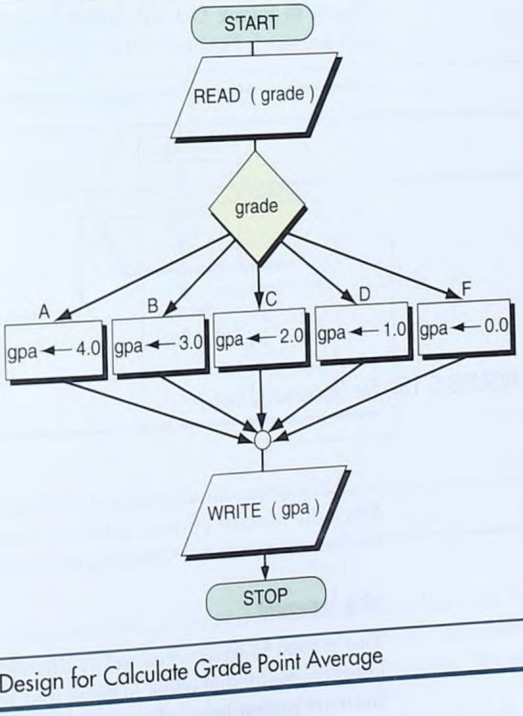


FIGURE C-11 Example: Design for Calculate Grade Point Average

As is the case in all structured programming constructs, the body of the loop can contain one and only one statement. As is the case with the other constructs, this one statement can be null or a compound statement. Figure C-12 shows the *for* construct.

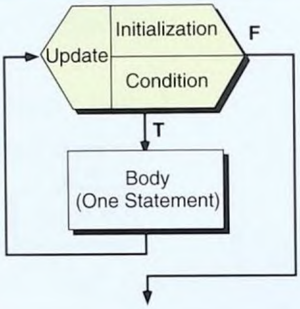
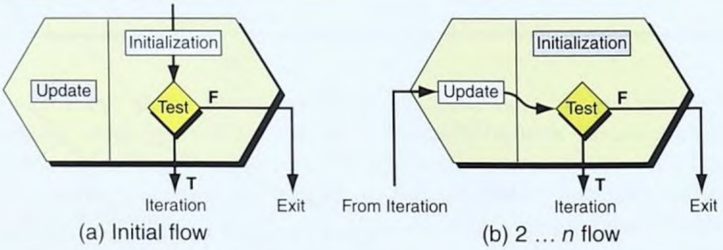


FIGURE C-12 *for* Statement

The *for* loop uses three actions: initialization, test, and update. However, only two of these actions are required in each iteration flow. In the first flow, when we enter the loop, only initialization and loop are used. This flow is shown in Figure C-13(a). In all iterations except the first, only update and test are used. Figure C-13(b) shows these flows.



(a) Initial flow

(b) 2 … *n* flow

FIGURE C-13 *for* Statement Flow

Let's design an algorithm that will read 20 numbers and print their sum. Since the number of times is known in advance, *for* is an excellent choice for the looping construct. The design for this program is shown in Figure C-14.

while Statement

The second looping construct is the *while* statement. The major difference between the *for* and *while* loops is that the *while* loop is not a counting loop. Both are pretest loops; therefore, like the *for,* the body of the *while* loop may never be executed.

We use the same basic symbol for the *while* loop, but since there is only a limit test, the internal divisions are not necessary. Figure C-15 shows the basic format of the *while* statement.

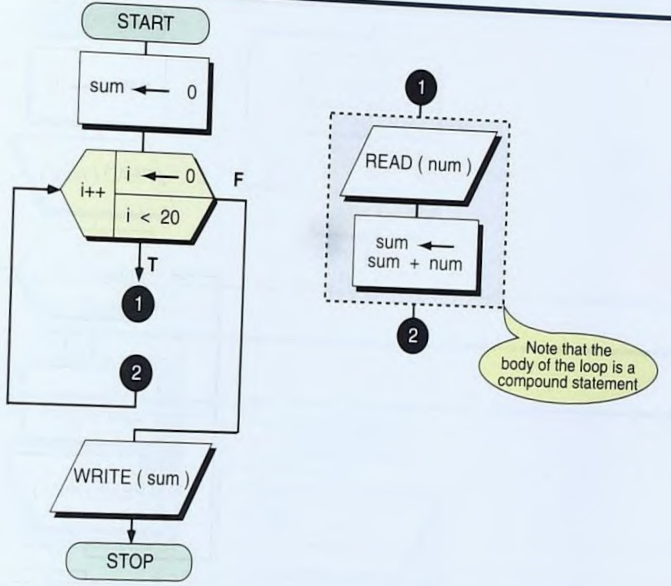


FIGURE C-14 Design to Read 20 Numbers

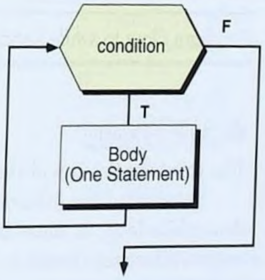


FIGURE C-15 *while* Statement Format

Let's design another program that reads numbers from the keyboard and prints their total. This time, we don't know how many numbers we may be reading. All we know is that all the numbers are positive. We can therefore signal the end of the numbers by having the user key –1.

Since we don't know how many times we will loop, we need a different construct; the *while* is well designed for this type of logic. The program flow is seen in Figure C-16. Note that in this design, the first number is read *before* the loop. This is known as priming the loop and is common in pretest loops.

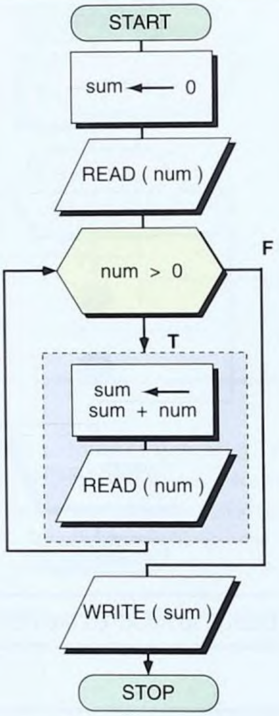


FIGURE C-16 Program Flow to *while* Read and Total

do...while Statement

The third application of the loop symbol is the *do...while* statement.

Because of the inherent differences between the *for* and *while* loops and *do...while* loop, it must be used differently in a flowchart. There are two major differences between the *while* and the *do...while*:

1. A *while* loop is a pretest loop. The *do...while* loop is a post-test loop.
2. The body of a *while* loop may never be executed. The body of a *do...while* loop is executed at least once.

Figure C-17 shows the use of the *do...while* statement. Note that in this statement, the condition is tested at the end of the loop.

Let's design an algorithm that reads and processes a number. In this algorithm, the number must be between 1 and 5. To make the program robust, we use a *do...while* loop that forces the user to enter a valid number. This is a common technique for validating user input. Note that as in previous looping constructs, the *do...while* loop can have only one statement. Therefore, we enclose the prompt and read in a composite symbol. The algorithm is shown in Figure C-18.

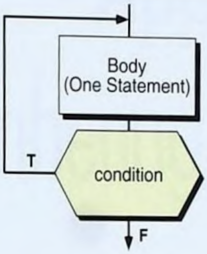


FIGURE C-17 *do…while* Statement

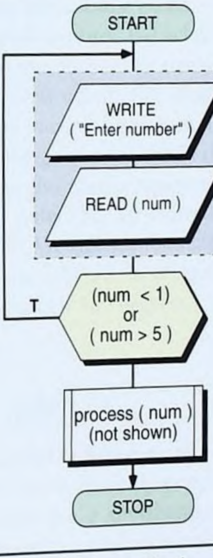


FIGURE C-18 Example: Algorithm for Input Validation with *do…while*

Appendix D Numbering Systems

Today the whole world uses the decimal number system developed by Arabian mathematicians in the eighth century. We acknowledge their contribution to numbers when we refer to our decimal system as Arabic numerals. But decimal numbers were not always commonly used. The first to use a decimal numbering system were the ancient Egyptians. The Babylonians improved on the Egyptian system by making the positions in the numbering systems meaningful. But the Babylonians also used a sexagesimal (base 60) numbering system. Whereas our decimal system has 10 values in its graphic representations, a sexagesimal system has 60. We still see remnants of the Babylonians' sexagesimal system in time, which is based on 60 minutes to an hour, and in the division of circles, which contain 360 degrees.

D.1 Computer Numbering System

Computer science uses several numbering systems. The computer itself uses binary (base 2). A binary system has only two values for each number position, 0 and 1. Programmers use a shorthand notation to represent binary numbers, hexadecimal (base 16). And of course, programmers also use the decimal system (base 10). Occasionally, we also encounter applications that require we use base 256. Since all these systems are used in C, we need to have a basic understanding of each to fully understand the language.

All of the numbering systems examined here are positional, meaning that the position of a symbol in relation to other symbols determines its value. Each symbol in a number has a position. In integrals and the integral portion of real numbers, the position starts from 0 and goes to $n - 1$, where n is the number of symbols in the integral part. In the fraction part of real numbers, the position starts from -1 and goes to $-m$, where m is the number of symbols in the fraction part. Each position is assigned a *weight*; the weights vary according to the numbering system.

Decimal Numbers (Base 10)

We all readily understand the **decimal numbers** (**base 10**). In fact, we have used it so much that it is intuitive. All of our terms for indicating countable quantities are based on it, and, in fact, when we speak of other numbering systems, we tend to refer to their quantities by their decimal equivalents.

The word *decimal* is derived from the Latin stem *deci,* meaning ten. The decimal system uses 10 symbols to represent quantitative values: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

Decimal numbers use 10 symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

For example, in Figure D-1, the decimal number 14782.721 has eight digits in positions –3 to 4.

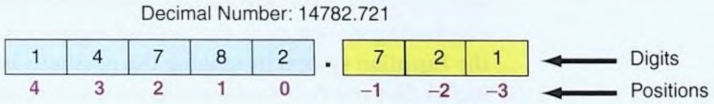


FIGURE D-1 The Decimal Number 14782.721

Weights

In the decimal system, each weight is 10 raised to the power of its position. The weight of the symbol at position –1 is 10^{-1} (1/10) while the weight of the symbol at position 0 is 10^{0} (1) and the weight of the symbol at position 1 is 10^{1} (10); and so on.

Binary Numbers (Base 2)

The **binary number** system (**base 2**) provides the basis for all computer operations. The binary system uses two symbols, 0 and 1. The word *binary* derives from the Latin stem *bi,* meaning two.

Binary numbers use two symbols: 0 and 1.

Weights

In the binary system, each weight equals 2 raised to the power of its position. The weight of the symbol at position –1 is 2^{-1} (1/2); the weight of the symbol at position 0 is 2^{0} (1); the weight of the symbol at position 1 is 2^{1} (2); and so on.

Binary-to-Decimal Conversion

To convert a binary number to decimal, we use the weights. We multiply each digit by its weight and add all of the weighted results. Figure D-2 shows how we can convert binary 1001110.101 to its decimal equivalent 78. 625.

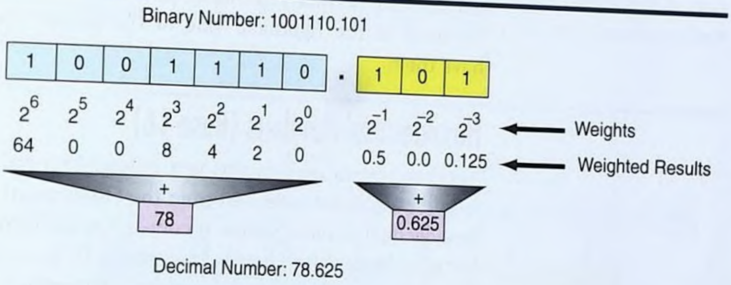


FIGURE D-2 Binary-to-Decimal Conversion

Decimal-to-Binary Conversion

Two simple operations, divide and multiply, give us a convenient way to convert a decimal number to its binary equivalent as shown in Figure D-3. To convert the integral part we divide the number by 2 and write down the remainder, which must be 0 or 1. The first remainder becomes the least significant binary digit. Now, we divide the quotient of that division by 2 and write down the new remainder in the second position. We repeat this process until the quotient becomes zero.

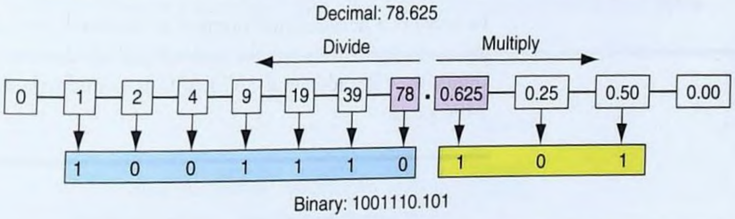


FIGURE D-3 Decimal-to-Binary Conversion

To convert the fractional part, we need to multiply the fractional part by two. The integral in the product is either a 0 or a 1; it becomes the binary digit. We then multiply the fractional part of the product by two to get the next binary digit and continue until the product is zero. For example, to convert 0.625 to binary, we multiply it by 2, resulting in 1.25. We take the integral part 1, and move the fraction part (0.25) to the next step. In the next step, after we multiply the 0.25 by two, we get 0.5. The integral part is 0, which we keep. The fractional part is 0.50, which we move to the next step. However, we need to limit the process because the product may never become zero. If the resulting fraction becomes 0.0, we stop because more binary digits does not contribute to the precision of the number. If the product does not become zero, we need to make a decision as to how many digits we need at the righthand side of the binary number and stop when we have them.

Hexadecimal Numbers (Base 16)

Another system used in this text is **hexadecimal numbers** (**base 16**). The word *hexadecimal* is derived from the Greek word *hexadec*, meaning 16. The hexadecimal number system is convenient for formatting a large binary number in a shorter form. It uses 16 symbols, 0, 1, ..., 9, A, B, C, D, E, and F. The hexadecimal system uses the same first 10 symbols as the decimal system, but instead of using 10, 11, 12, 13, 14, and 15, it uses A, B, C, D, E, and F. This prevents any confusion between two adjacent symbols. Note that the hexadecimal symbols A to F can be either upper- or lowercase.

Hexadecimal numbers use 16 symbols: 0, 1,..., 9, A, B, C, D, E, and F.

Weights

In the hexadecimal system, each weight equals 16 raised to the power of its position. The weight of the symbol at position 0 is 16^0 (1); the weight of the symbol at position 1 is 16^1 (16); and so on.

Hexadecimal-to-Decimal Conversion

To convert a hexadecimal number to decimal, we use the weights. We multiply each digit by its weight and add all of the weighted results. Figure D-4 shows how hexadecimal 3A73.A0C is transformed to its decimal equivalent 14963.628.

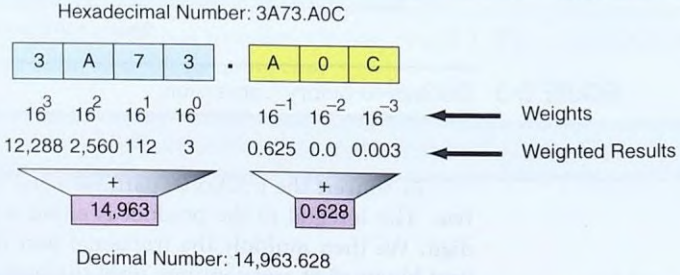


FIGURE D-4 Hexadecimal-to-Decimal Conversion

Decimal-to-Hexadecimal Conversion

We use the same process we used for changing decimal to binary to transform a decimal number to hexadecimal. The only difference is that we divide the number by 16 instead of 2 to get the integral part and we multiply the number by 16 to get the fractional part. Figure D-5 shows how 14963.628 in decimal is converted to hexadecimal 3A73.A0C. Note that we stop after three digits in the fractional part.

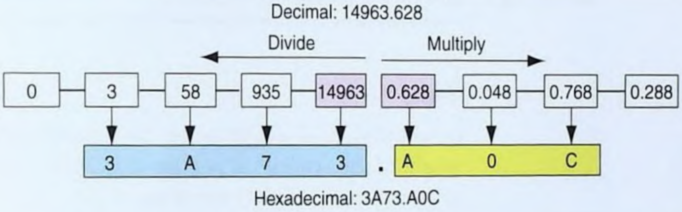


FIGURE D-5 Decimal-to-Hexadecimal Conversion

Base 256

Another numbering system that is used in computer science is **base 256**. We encounter this base normally in two situations: when we need to make a number from individual bytes and when we are dealing with Internet addresses.

In the first application, we normally have individual bytes where each byte contains an unsigned number between 0 to 255. We need to consider several of these bytes as a number. Figure D-6 shows the situation with four bytes.

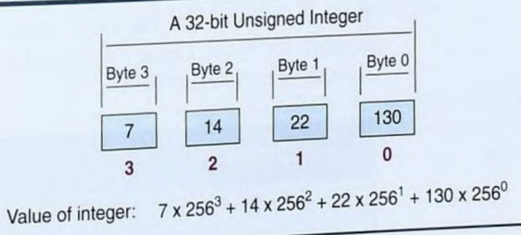

A 32-bit Unsigned Integer
Byte 3
Byte 2
Byte 1
Byte 0
7
14
22
130
3
2
1
0
Value of integer: 7 x 256³ + 14 x 256² + 22 x 256¹ + 130 x 256⁰

FIGURE D-6 Byte Conversion

Another application is in Internet addresses. Internet addresses in version 4 use base 256 to represent an address in dotted-decimal notation. When we define an address as 131.32.7.8, we are using a base-256 number. To distinguish between the numbers, a *dot* is used. For example, the address

131.32.7.8 is made of four numbers 8, 7, 32, and 131 at positions 0, 1, 2, 3, respectively, as shown in Figure D-7.

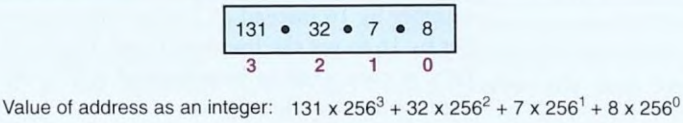

131 • 32 • 7 • 8
3
2
1
0
Value of address as an integer: $131 \times 256^3 + 32 \times 256^2 + 7 \times 256^1 + 8 \times 256^0$

FIGURE D-7 Internet Dotted-Decimal Notation

Weights

In base 256, each weight equals 256 raised to the power of its position. The weight of the symbol[1] at position 0 is 256^0 (1); the weight of the symbol at position 1 is 256^1 (256); and so on.

Base 256-to-Decimal Conversion

To convert a base 256 number to its decimal equivalent, we follow the same process we discussed for converting base 2 or base 16 to decimal; the weights, however, are 256^n where n is the position. Note, however, that in this case, we are normally dealing with an integral number.

Decimal-to-Base 256 Conversion

To convert a decimal number to base 256, we follow the same method we discussed for converting decimal to base 2 or base 16, but we divide the number by 256 and keep the remainder.

A Comparison

Table D-1 shows how the three systems represent the decimal numbers 0 through 15. As you can see, decimal 13 is equivalent to binary 1101, which is equivalent to hexadecimal D.

Decimal	Binary	Hexadecimal
0	0	0
1	1	1
2	10	2

continued

TABLE D-1 Comparison of Decimal, Binary, and Hexadecimal Systems

[1] In base 256, each symbol can be one to three digits.

Decimal	Binary	Hexadecimal
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

TABLE D-1 Comparison of Decimal, Binary, and Hexadecimal Systems *(continued)*

Other Conversions

There are other conversions such as base 2 to base 16 or base 16 to base 256. For these conversions, we use base 10 as the intermediate system. In other words, to change a number from binary to hexadecimal we first change the binary to decimal and then change the decimal to hexadecimal. We discuss some easy methods for common conversions.

Binary-to-Hexadecimal Conversion

To change a number from binary to hexadecimal, we group the binary digits from the right by fours. Then we convert each four-bit group to its hexadecimal equivalent using Table D-1. In Figure D-8, we convert binary 1010001110 to hexadecimal.

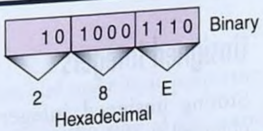


FIGURE D-8 Binary-to-Hexadecimal Conversion

Hexadecimal-to-Binary Conversion

To change a hexadecimal number to binary, we convert each hexadecimal digit to its equivalent binary number using Table D-1 and concatenating the results. In Figure D-9, we convert hexadecimal 28E to binary.

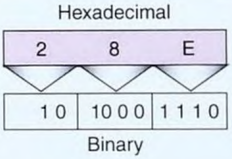


FIGURE D-9 Hexadecimal-to-Binary Conversion

Base 256-to-Binary Conversion

To convert a base 256 number to binary, we first need to convert the number in each position to an 8-bit binary group and then concatenate the groups.

Binary-to-Base 256 Conversion

To convert from binary to base 256, we need to divide the binary number into groups of 8 bits, convert each group to decimal, and then insert separators (dots) between the decimal numbers.

D.2 Storing Integers

We have discussed how integers and real numbers are presented in different bases. Although base 16 and base 256 are used in computer science, data are stored in the computer in binary. Numbers must be changed to base 2 to be stored in the computer. All of our discussions so far have ignored the sign of the number. In computer science, we use both positive and negative numbers. We need some way to store the sign. In this section, we concentrate on how integers are stored in a computer; in the next section, we show how real numbers are stored.

Unsigned Integers

Storing **unsigned integers** is a straightforward process. The number is changed to the corresponding binary form, and the binary representation is stored. For example, an unsigned integer can be stored as a number from 0 to 15 in a 4-bit integer, as shown in Figure D-10.

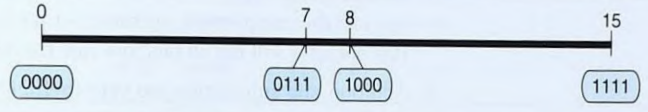


FIGURE D-10 Unsigned Integers Format

Addition of unsigned integers is very straightforward as long as there is no overflow (see the section on overflow). Subtraction of unsigned integers must be done with caution. If the result is negative, the number is not an unsigned number anymore. Normally computers promote the result to a signed integer in this case.

Signed Integers

Storing **signed integers** is different from storing unsigned integers because we must consider both positive and negative numbers. Four methods are designed to store signed integers in a computer: sign and magnitude, one's complement, two's complement, and Excess system.

Sign and Magnitude

In **sign and magnitude**, we divide the available range between the positive and negative numbers. The lower part of the range occupies the positive numbers; the upper part occupies the negative numbers. To do so, we consider the leftmost bit to represent the sign of the number and the rest of the bit to be the absolute value, the magnitude of the number. Let us see how a 4-bit integer can both hold the positive and negative numbers. Figure D-11 shows the partition of the range between positive and negative numbers.

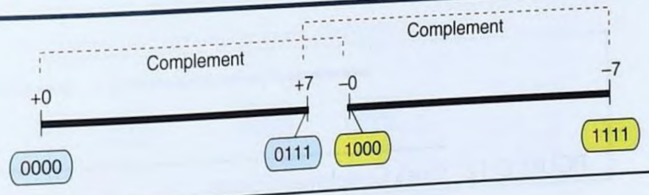


FIGURE D-11 Signed Integer Format

Properties

Let us summarize the sign and magnitude properties:

1. The leftmost bit contains the sign of the number; 0 for positive, 1 for negative.
2. There are two zeros in this method: +0 (0000) and –0 (1000).

3. To change the sign of a number, we need to flip only the first bit. We cannot use the complement operator (~) we learned in Chapter 14 because this operator will flip all bits, not just the leftmost bit.
4. Addition and subtraction are very inefficient operations. We need to do the following:
   a. Separate the sign from the magnitude.
   b. Compare two magnitudes to see if we need to add or subtract the magnitude.
   c. We have eight cases in total, four for addition and four for subtraction.
   d. Insert the sign of the result after the absolute value of the result is determined.

Applications

Properties 2, 3, and 4 make this method unsuitable for a general-purpose computer in which we need to perform mathematical operations. The sign and magnitude method, however, is used in computer science when we do not need mathematical operations. One of these areas is storing analog and digital signals.

One's Complement

One's complement is similar to sign and magnitude; however, the partition of the range between the positive and negative numbers is different. The numbers are arranged symmetrically. A positive and a negative number are symmetric with respect to the middle of the range. Figure D-12 shows how the positive and negative numbers are distributed.

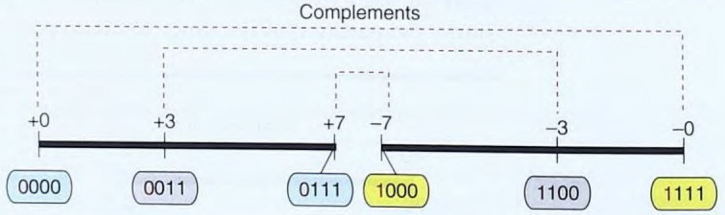


FIGURE D-12 One's Complement Format

Properties

Let us summarize the one's complement properties:

1. The leftmost bit contains the sign; 0 for positive, 1 for negative.
2. There are two zeros in this method: +0 (0000) and –0 (1111).
3. To flip the sign of a number, we flip each individual bit. We use the complement operator (~) we learned in Chapter 14 to find –A from A as shown in Figure D-13.

$$-3 \longrightarrow \sim(0\,0\,1\,1) \longrightarrow 1\,1\,0\,0$$
$$-(-3) \longrightarrow \sim(1\,1\,0\,0) \longrightarrow 0\,0\,1\,1$$

FIGURE D-13 Flipping Signs in One's Complement

The figure also shows that if we complement –A, we get back A. In one's complement, we also can find the complement of A as $-A = (2^n - 1) - A$. For example, –3 is stored as 15 –3 or 12, when n is 4 (number of bits in the integer).

4. If we add A + (–A) we get –0 (all bits are 1s). We saw this property in Chapter 14 when we used checksum.

5. Addition and subtraction are very simple operations. To add (A + B), we just add the numbers bit by bit. To subtract A – B, we just add A and the complement of B. In other words, A – B = A + (– B). The only thing we need to consider about adding or subtracting is to add the carry produced at the last column to the result. Figure D-14 shows how numbers are complemented, added, and subtracted.

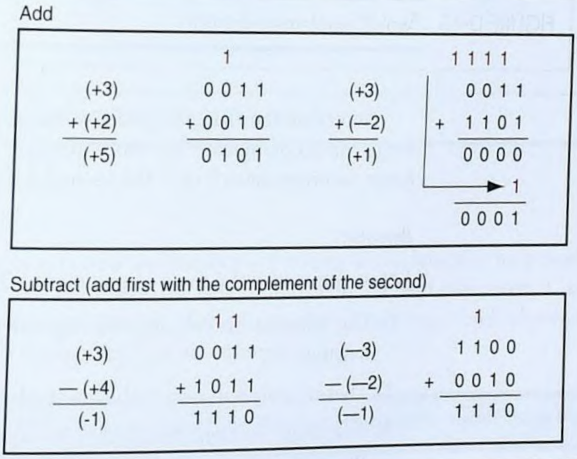


FIGURE D-14 Add and Subtract One's Complement

Applications

Early general-purpose computers, such as VAX, used one's complement. The modern general-purpose computer does not use this method for arithmetic calculation because of the second property (existence of two 0s) and the need for keeping track of the carry from the last column. However, one's complement arithmetic has its own place in computer science. We saw the use of it in the checksum calculation in Chapter 14. We simulated the use of one's complement with unsigned numbers. To simulate adding the carry from the last column to the partial result, we used modulo $2n - 1$ in our calculations.

Two's Complement

Like one's complement, **two's complement** also shares the range between positive and negative numbers. However, the partition of the range between the positive and negative numbers is different. The symmetry is an offset. For example, the fourth number from the beginning of the range is the complement of the third number from the end. This is done for two purposes: to eliminate the negative 0 and to avoid adding the carry from the last column to the partial result. Figure D-15 shows how the positive and negative numbers are distributed.

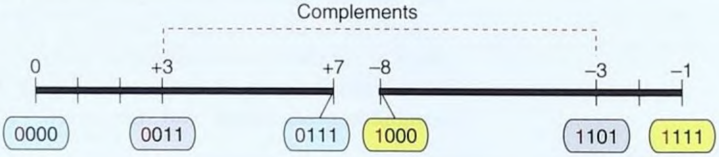


FIGURE D-15 Two's Complement Format

Note that the 0 in the positive range, which means that the negative range can accommodate one more number. In the figure, the first half of the range accommodates 0 to 7; the second half accommodates −1 to −8.

Properties

Let us summarize the two's complement properties:

1. The leftmost bit contains the sign of the number; 0 for positive, 1 for negative.
2. There is only one zero in this method: 0 (0000).
3. To change the sign of a number, we need two operations. First we need to flip each individual bit, then we need to add 1 to the previous result as shown in Figure D-16.

$$-3 \longrightarrow \sim(0\,0\,1\,1) + 1 \longrightarrow 1\,1\,0\,1$$

$$-(-3) \longrightarrow \sim(1\,1\,0\,0) + 1 \longrightarrow 0\,0\,1\,1$$

FIGURE D-16 Complementing Two's Complement

The figure also shows that if we complement –A, we get back A. In two's complement, we also can find the complement of A as –A = (2^n) – A, where n is the number of bits in the integer. For example, –3 is stored as 16 –3 or 13.

4. If we add A + (–A) we get 0.
5. Addition and subtraction are simpler. To add (A + B), we just add the numbers bit by bit. To subtract (A – B), we just add A with the complement of B. We do not have to worry about the carry from the last column; we drop it. Figure D-17 shows this feature.

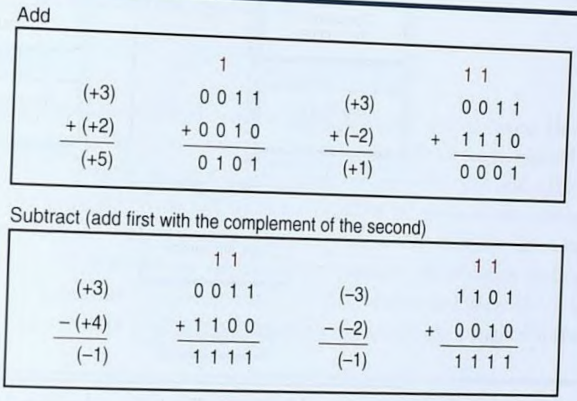


FIGURE D-17 Adding and Subtracting Two's Complement

Applications

One single zero and the simplicity of adding and subtracting have made two's complement arithmetic the best candidate for modern computers. C uses the hardware instruction set of the computers it runs on, which almost universally uses two's complement arithmetic.

Modern computers use two's complement to store signed integers.

Storing and Retrieving Two's Complement

As we said before, computers today use two's complement numbers for handling mathematical problems. However, we may wonder how an input function, such as *scanf*, converts a signed decimal integer to two's complement, and an output function, such as *printf*, converts a two's complement number to a signed decimal integer. Figure D-18 shows two high-level algorithms that can be used by these functions.

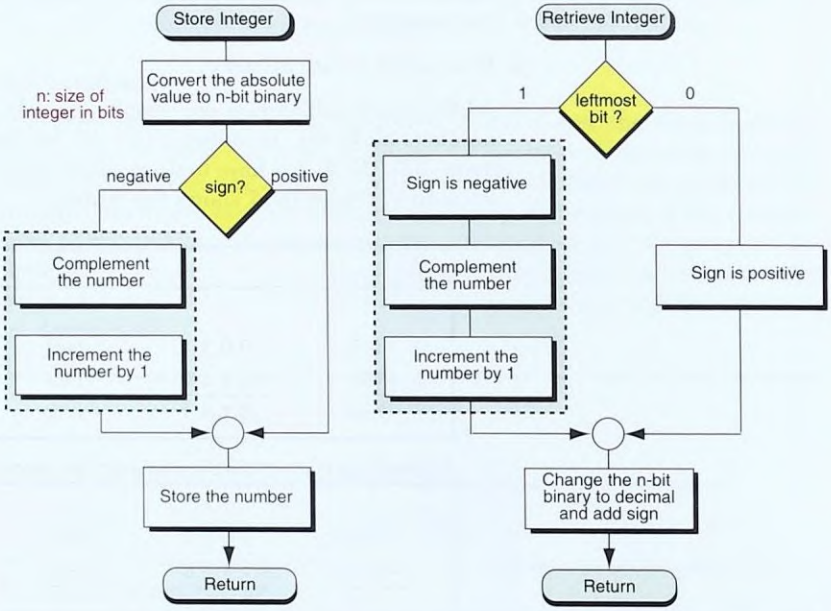


FIGURE D-18 Storing and Retrieving Two's Complement

EXAMPLE D-1 Let's follow the storing algorithm to see how +76 is stored in a 16-bit integer. The absolute value of the number (76) is changed to a 16-bit binary number. The sign is positive, so the number is stored in the memory.

```
Decimal number:                +76
Convert absolute value:        0000000001001100
Store value:                   0000000001001100
```

EXAMPLE D-2 Let's follow the storing algorithm to see how −76 is stored in a 16-bit integer. The absolute value of the number (76) is changed to a 16-bit binary number. The sign is negative, so we need to complement the number and add 1 to it as shown below.

```
Decimal number:                -76
Convert absolute value:        0000000001001100
Complement:                    1111111110110011
Add  1:                        1111111110110100
Store value:                   1111111110110100
```

EXAMPLE D-3 Let's follow the retrieving algorithm to see how a stored value in a 16-bit integer is retrieved. The process is shown below.

```
Retrieved value:     1111111111101011 (sign "-")
Complement:          0000000000010100
Add 1:               0000000000010101
Convert to decimal:  21
Add sign:            -21
```

The retrieved binary value is negative, so the sign is stored to be added at the end. The number is complemented and 1 is added. The result is converted to decimal and the sign is added.

Excess

There are applications that require more numeric comparisons than arithmetic operations. In these cases we can use a simple strategy, named **Excess** by the IEEE, to store positive and negative integers. We just add a fixed value (called the bias value) to the negative number to make them non-negative (zero or positive) when we store the numbers. We subtract the same bias value when we retrieve the number. For example, in our hypothetical 4-bit integer, we can store numbers from –7 to 8 as shown in Figure D-19. In other words, we store both positive and negative numbers in an unsigned format.

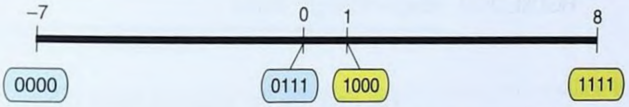


FIGURE D-19 Excess Format

When two numbers are stored using the Excess method, we can easily compare them on the value; we do not have to worry about the sign of the numbers. When adding two numbers in the Excess system, the bias value is added twice; therefore, the bias value must be subtracted when we store the result. When subtracting two numbers, the bias value is cancelled during the subtraction; therefore, it must be added back when we store the result.

Overflow

We need to discuss a very important issue, overflow. Overflow can be the source of much confusion for a programmer. Sometimes when we print a number, we get a surprising result. Often the reason is that an overflow has occurred. Overflow occurs because integers are stored on a limited size word in the computer. For example, if the size of the integer is only 4 bits, we can either store an unsigned integer between 0 and 15 or a signed integer between −8 and +7 (using two's complement). Any number beyond this range overflows the possible values. Most of the time, the system does not issue an error or warning, it just drops the extra bit or bits that do not fit in the allocated space. This creates an invalid result, which may be positive or negative. Then when we print the results, we get a surprising number.

Overflow is better understood if we show the range of the integers that can be stored in a number in a circle. Figure D-20 shows the range for two methods that are used in today's computers, unsigned integer and two's complement.

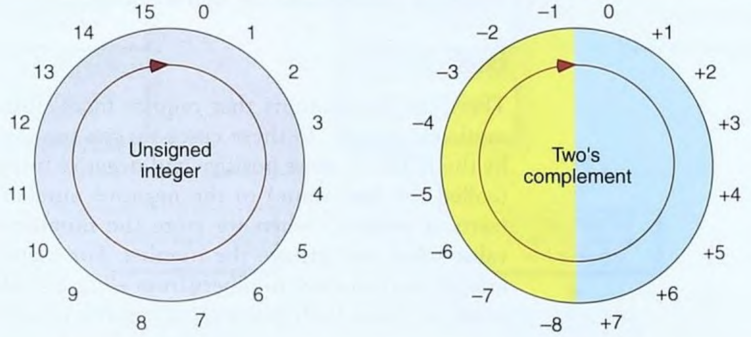


FIGURE D-20 Range of Integer Values

The circle for the unsigned integer shows that 15 + 1 is 0. The circle for two's complement shows that 7 + 1 is −8. If we increment an unsigned value holding the maximum possible number, we get 0. If we increment an integer value holding the maximum possible value, we get the minimum possible value. We cannot test this concept with 4-bit integers, but we can prove it with the actual size of the integer in any computer as shown in Program D-1.

PROGRAM D-1 Demonstrate Overflow

```
/* Demonstrate circular nature of unsigned and two's
   complement integer numbers.
      Written by:
      Date:
*/
#include <stdio.h>
#include <limits.h>

int main (void)
{
```

continued

PROGRAM D-1 Demonstrate Overflow *(continued)*

```
// Local Definitions
   unsigned short x = USHRT_MAX;
            short y = SHRT_MAX;

// Statements
   printf("Maximum unsigned short value:     %u\n", x);
   x++;
   printf("Maximum unsigned short value + 1: %u\n", x);

   printf("Maximum short value:              %d\n", y);
   y++;
   printf("Maximum short value + 1 :         %d\n", y);

   return 0;
}  // main
```

```
Results:
Maximum unsigned value:     65535
Maximum unsigned value + 1: 0
Maximum short value:        32767
Maximum short value + 1 :   -32768
```

D.3 Storing Real Numbers

We are all familiar with scientific notation to represent a real number. In this notation, we can show a real number such as –314.625 as $-3.14625 \times 10^{+2}$. We also know that a real number is made of three pieces of information: the sign (–), the precision (3.14625), and the power of ten (+2). Note that we do not have to show the multiplication operator or the base of the power (10) because they are understood by the rules of scientific notation.

Computers use the scientific notation concepts to store a real number. If we write all information in binary system, 314.625 is represented as shown below with the power also in binary (4).

$$-10011.10101010 \times 2^{100}$$

Normalization

We have one problem to solve before we can store this number in the computer: the position of the binary point (the point that separates the integral part from the fractional part). We can store only binary digits, not a point. The solution is **normalization**. We normalize the number so that the point is always at a fixed position. Tradition and the standard state that the number should have only one binary digit to the left of the point. For non-zero values, the digit is a 1. To normalize the number, therefore, we shift the point to the left or to the right based on its original position. Shifting binary numbers requires multiplying or dividing the number by two for each shift. In other words, if we move the point to the left, we need to add the number of digits shifted to power; if we move the point to the right, we need to subtract the number of digits shifted from the power. The normalized version of the previous number is shown below. Note that we moved the point four positions to the right, and we added 4 to the power, making it 8 (the power is represented in binary).

$$-1.001110101010 \times 2^{1000}$$

After normalization, we are left with only the sign, the precision, and the power to store; we do not have to store the integral part of the precision or the binary point.

Fortunately, C99 provides us with help to find how real numbers are stored in the computer. We can use the %A or %a conversion code to print the real number in a format that represents these values. Program D-2 displays a positive and a negative real number. Note that the values are the same except for the sign.

PROGRAM D-2 Demonstrate Storage of Real Numbers

```
/* Demonstrate the storage of real numbers.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declaration
   float x = -314.625;
   float y = +314.625;

// Statements
   printf ("-314.625: %A\n", x);
   printf ("+314.625: %A\n", y);
   return 0;
} // main
```

```
Results:
-314.625: -0X1.3AA0000000000P+8
+314.625:  0X1.3AA0000000000P+8
```

The results show the sign, the mantissa (3AA0000000000 in hexadecimal), and the power (8).

Sign, Exponent, and Mantissa

Real numbers contain three parts, which are referred to as the sign (s), exponent, and mantissa (m). We can say that the original number (N) is

$$N = (-1)^s \times 1.m \times 2^e$$

Sign

The **sign** of the number is stored using one bit (0 for plus and 1 for minus).

Exponent

The **exponent** (power of 2) defines the power. Note that the power can be negative or positive. Excess is the method used to store the exponent.

Mantissa

The **mantissa** is the binary number to the right of the binary point. It defines the precision of the number. The mantissa is stored as an unsigned integer.

IEEE Standards

The Institute of Electrical and Electronics Engineers (IEEE) has defined two standards to store numbers in memory (single precision and double precision). These formats are shown in Figure D-21. Note that the number inside the boxes is the number of bits for each field.

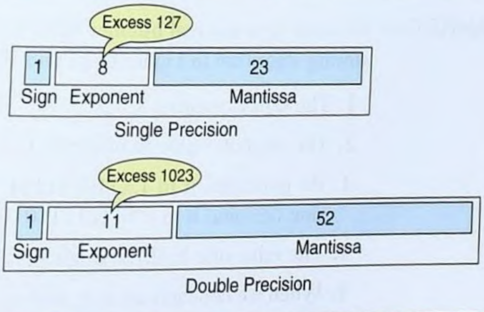


FIGURE D-21 IEEE Standards for Floating-Point Representation

Storing and Retrieving Algorithm

Let's look at the algorithms for storing and retrieving real numbers. These algorithms, shown in Figure D-22, give us an insight into how the input and output functions such as *scanf* and *printf* store or retrieve real numbers.

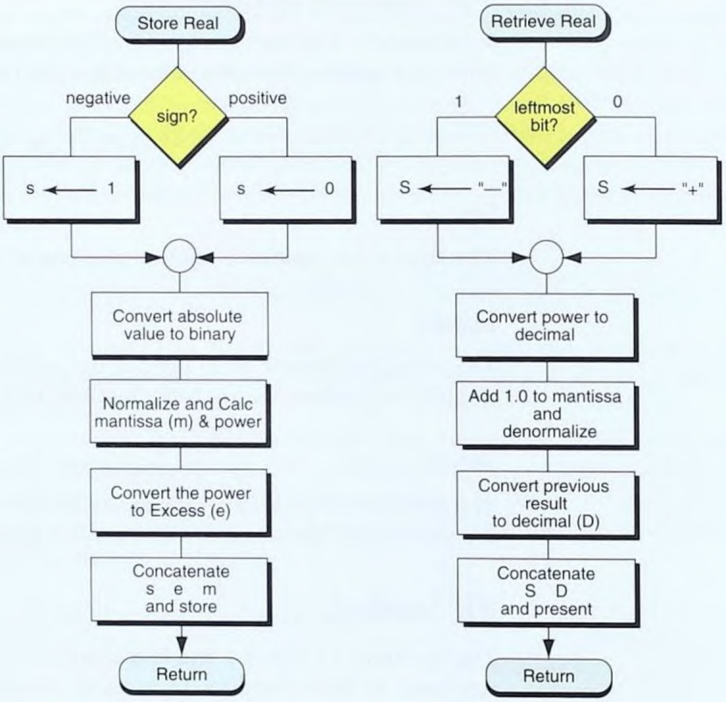

Store Real
sign?
negative
positive
s ← 1
s ← 0
Convert absolute value to binary
Normalize and Calc mantissa (m) & power
Convert the power to Excess (e)
Concatenate s e m and store
Return
Retrieve Real
leftmost bit?
1
0
S ← "–"
S ← "+"
Convert power to decimal
Add 1.0 to mantissa and denormalize
Convert previous result to decimal (D)
Concatenate S D and present
Return

FIGURE D-22 IEEE Algorithms

EXAMPLE D-4 We show how the real number 123.8125 is stored in the computer using the storing algorithm in Figure D-22 and IEEE for single precision.

1. The sign is positive; s = 0.
2. The absolute value in binary is 1111011.1101
3. We normalize it to 1.1110111101. The value of m in IEEE single precision (23 bits) is m = 11101111010000000000000 and the power is 6.
4. The value of e in IEEE single precision is 6 +127 or 10000101.
5. When we concatenate s, e, and m, we have the value shown below:

```
0 10000101 11101111010000000000000
```

EXAMPLE D-5 We show how to find the decimal value of the following 32-bit real number stored using IEEE single precision.

```
1 10000010 00011000000000000000000
```

We use the retrieve algorithm in Figure D-22.

1. The leftmost bit is 1; S = "–".
2. The value of next 8 bits is 130. We subtract 127 from it to get 3.
3. We add 1 and the binary point to m and shift the binary point 3 digits to the right to get 1000.11 (ignoring the trailing zeros).
4. We convert the above number to decimal to get D = 8.75.
5. We concatenate S and D to get –8.75.

Appendix E Integer and Float Libraries

This appendix documents two of the more important C libraries. Note that the values shown here are representative only and change from hardware to hardware. Libraries contain unformatted numbers (no commas), often expressed in hexadecimal. We use decimally formatted numbers for readability.

E.1 limits.h

Table E-1 contains hardware-specific values for the integer types.

Identifier	Meaning	Minimum Value
CHAR_BIT	bits in a char	8
SCHAR_MIN	short char minimum	–127
SCHAR_MAX	short char maximum	127
UCHAR_MAX	unsigned char maximum	255
CHAR_MIN	char minimum	See SCHAR_MIN
CHAR_MAX	char maximum	See SCHAR_MAX
SHRT_MIN	short int minimum	–32,767
SHRT_MAX	short int maximum	32,767
USHRT_MAX	unsigned short maximum	65,535
INT_MIN	int minimum	–32,767
INT_MAX	int maximum	32,767

continued

TABLE E-1 Partial Contents of Limits Library

Identifier	Meaning	Minimum Value
UINT_MAX	unsigned int maximum	65,535
LONG_MIN	long minimum	–2,147,483,647
LONG_MAX	long maximum	2,147,483,647
ULONG_MAX	unsigned long maximum	4,294,967,295
LLONG_MIN	long long int minimum	$-(2^{63} - 1)$
LLONG_MAX	long long int maximum	$2^{63} - 1$
ULLONG_MAX	unsigned long long maximum	$2^{64} - 1$

TABLE E-1 Partial Contents of Limits Library *(continued)*

E.2 float.h

Table E-2 contains hardware-specific values for the floating-point types.

Identifier	Meaning	Minimum Value
FLT_DIG	digits of precision	6
DBL_DIG		10
LDBL_DIG		10
DECIMAL_DIG	decimal digits needed to represent floating-point value	10
FLT_MANT_DIG	size of mantissa	none
DBL_MANT_DIG		none
LDBL_MANT_DIG		none
FLT_MIN_EXP	smallest integer for negative exponent (float radix)	none
DBL_MIN_EXP		none
LDBL_MIN_EXP		none
FLT_MIN_10_EXP	smallest integer for negative exponent (base 10)	–37
DBL_MIN_10_EXP		–37
LDBL_MIN_10_EXP		–37

continued

TABLE E-2 Partial Contents of Limits Library

Identifier	Meaning	Minimum Value
FLT_MAX_EXP	largest integer for positive exponent (float radix)	37
DBL_MAX_EXP		37
LDBL_MAX_EXP		37
FLT_MAX_10_EXP	largest integer for positive exponent (base 10)	37
DBL_MAX_10_EXP		37
LDBL_MAX_10_EXP		37
FLT_MAX	largest possible floating–point number	10^{37}
DBL_MAX		10^{37}
LDBL_MAX		10^{37}
FLT_MIN	smallest possible floating–point number	10^{-37}
DBL_MIN		10^{-37}
LDBL_MIN		10^{-37}

TABLE E-2 Partial Contents of Limits Library *(continued)*

Appendix F
Function Libraries

In this appendix we list most of the standard functions found in the C language. We have grouped them by library so that related functions are grouped together. We have also listed them alphabetically in Section F.1 for your convenience. Note that not all functions are covered in the text and there are functions in the libraries that are not covered in this appendix.

F.1 Function Index

Function	Page	Library	Function	Page	Library	Function	Page	Library
_Exit	1068	stdlib	atanl	1063	math	ceill	1063	math
abort	1068	stdlib	atan2	1063	math	clearerr	1066	stdio
abs	1067	stdlib	atan2f	1063	math	clock	1069	time
acos	1063	math	atan2l	1063	math	cos	1063	math
aosf	1063	math	atexit	1068	stdlib	cosf	1063	math
acosl	1063	math	atof	1068	stdlib	cosl	1063	math
asctime	1069	time	atoi	1068	stdlib	cosh	1063	math
asin	1063	math	atol	1068	stdlib	coshf	1063	math
asinf	1063	math	atoll	1068	stdlib	coshl	1063	math
asinl	1063	math	calloc	1068	stdlib	ctime	1069	time
atan	1063	math	ceil	1063	math	difftime	1069	time
atanf	1063	math	ceilf	1063	math	div	1067	stdlib

continued

Function	Page	Library	Function	Page	Library	Function	Page	Library
atanl	1063	math	isupper	1063	ctype	scaleblnf	1065	math
atan2	1063	math	isxdigit	1063	ctype	scaleblnl	1065	math
atan2f	1063	math	iswalnum	1063	wctype_t	scanf	1066	stdio
atan2l	1063	math	iswalpha	1063	wctype_t	sin	1065	math
atexit	1068	stdlib	iswcntrl	1063	wctype_t	sinf	1065	math
atof	1068	stdlib	iswdigit	1063	wctype_t	sinl	1065	math
atoi	1068	stdlib	iswgraph	1063	wctype_t	sinh	1065	math
atol	1068	stdlib	iswlower	1063	wctype_t	sinhf	1065	math
atoll	1068	stdlib	iswprint	1063	wctype_t	sinhl	1065	math
calloc	1068	stdlib	iswpunct	1063	wctype_t	snprintf	1066	stdio
ceil	1063	math	iswspace	1063	wctype_t	sprintf	1066	stdio
ceilf	1063	math	iswupper	1063	wctype_t	swprintf	1066	stdio
ceill	1063	math	iswxdigit	1063	wctype_t	sqrt	1065	math
clearerr	1066	stdio	labs	1067	stdlib	sqrtf	1065	math
clock	1069	time	llabs	1067	stdlib	sqrtl	1065	math
cos	1063	math	ldexp	1064	math	srand	1067	stdlib
cosf	1063	math	ldexpf	1064	math	sscanf	1066	stdio
cosl	1063	math	ldexpl	1064	math	swscanf	1066	stdio
cosh	1063	math	ldiv	1067	stdlib	strcat	1069	string
coshf	1063	math	lldiv	1067	stdlib	strchr	1069	string
coshl	1063	math	lint	1064	math	strcmp	1069	string
ctime	1069	time	lintf	1064	math	strcpy	1069	string
difftime	1069	time	lintl	1064	math	strcspn	1069	string
div	1067	stdlib	lrint	1064	math	strftime	1069	time
exit	1068	stdlib	lrintf	1064	math	strlen	1069	string
_Exit	1068	stdlib	lrintl	1064	math	strncat	1069	string
exp	1063	math	llrint	1064	math	strncmp	1069	string
expf	1064	math	llrintf	1064	math	strncpy	1069	string
expl	1064	math	llrintl	1064	math	strpbrk	1069	string

continued

Function	Page	Library	Function	Page	Library	Function	Page	Library
expm1	1064	math	localtime	1069	time	strrchr	1069	string
expm1f	1064	math	log	1064	math	strspn	1069	string
expm1l	1064	math	logf	1064	math	strstr	1069	string
exp2	1064	math	logl	1064	math	strtod	1068	stdlib
exp2f	1064	math	log2	1064	math	strtof	1068	stdlib
exp2l	1064	math	log2f	1064	math	strtok	1069	string
fabs	1064	math	log2l	1064	math	strtol	1068	stdlib
fabsf	1064	math	log10	1064	math	strtold	1068	stdlib
fabsl	1064	math	log10f	1064	math	strtoll	1068	stdlib
fclose	1066	stdio	log10l	1064	math	strtoul	1068	stdlib
feof	1066	stdio	lround	1064	math	system	1068	stdlib
ferror	1066	stdio	lroundf	1064	math	tan	1065	math
fgetc	1066	stdio	lroundl	1064	math	tanf	1065	math
fgetwc	1066	stdio	llround	1064	math	tanl	1065	math
fgets	1067	stdio	llroundf	1064	math	tanh	1065	math
fgetws	1067	stdio	llroundl	1064	math	tanhf	1065	math
floor	1064	math	malloc	1068	stdlib	tanhl	1065	math
floorf	1064	math	memchr	1068	string	time	1069	time
floorl	1064	math	memcmp	1068	string	tmpfile	1067	stdio
fmod	1064	math	memcpy	1068	string	tmpnam	1067	stdio
fmodf	1064	math	memmove	1068	string	trunc	1065	math
fmodl	1064	math	mktime	1069	time	truncf	1065	math
fopen	1066	stdio	modf	1064	math	truncl	1065	math
freopen	1066	stdio	modff	1065	math	tolower	1063	ctype
fprintf	1066	stdio	modfl	1065	math	toupper	1063	ctype
fputc	1066	stdio	nearbyint	1065	math	towlower	1063	ctype
fputwc	1066	stdio	nearbyintf	1065	math	towupper	1063	ctype
fputs	1067	stdio	nearbyintl	1065	math	ungetc	1066	stdio
fputws	1067	stdio	pow	1065	math	ungetwc	1066	stdio

continued

Function	Page	Library	Function	Page	Library	Function	Page	Library
fread	1067	stdio	powf	1065	math	wcscat	1069	wchar
free	1068	stdlib	powl	1065	math	wcschr	1069	wchar
frexp	1064	math	printf	1066	stdio	wcsrchr	1069	wchar
frexpf	1064	math	putc	1066	stdio	wcscmp	1069	wchar
frexpl	1064	math	putwc	1066	stdio	wcscpy	1069	wchar
fscanf	1066	stdio	putchar	1066	stdio	wcscspn	1069	wchar
fwscanf	1066	stdio	putwchar	1066	stdio	wcslen	1069	string
fseek	1067	stdio	puts	1067	stdio	wcsncat	1069	wchar
ftell	1067	stdio	rand	1067	stdlib	wcsncmp	1069	wchar
fwprintf	1066	stdio	realloc	1068	stdlib	wcsncpy	1069	wchar
fwrite	1067	stdio	remainder	1065	math	wcspbrk	1069	wchar
getc	1066	stdio	remainderf	1065	math	wcsspn	1069	wchar
getchar	1066	stdio	remainderl	1065	math	wcsstr	1069	wchar
getwc	1066	stdio	remquo	1065	math	wcstok	1069	wchar
getwchar	1066	stdio	remquof	1065	math	wprintf	1066	stdio
gets	1067	stdio	remquol	1065	math	wscanf	1066	stdio
gmtime	1069	time	remove	1067	stdio			
ilogb	1064	math						

F.2 Type Library

The following functions are found in *ctype.h*.

isalnum	int	isalnum	(int a_char);
isalpha	int	isalpha	(int a_char);
isascii	int	isascii	(int a_char);[1]
iscntrl	int	iscntrl	(int a_char);
isdigit	int	isdigit	(int a_char);
isgraph	int	isgraph	(int a_char);
islower	int	islower	(int a_char);
isprint	int	isprint	(int a_char);

[1] Not standard C. Traditional extension included in most implementations.

ispunct	int	ispunct	(int a_char);
isspace	int	isspace	(int a_char);
isupper	int	isupper	(int a_char);
isxdigit	int	isxdigit	(int a_char);
tolower	int	tolower	(int a_char);
toupper	int	toupper	(int a_char);

The following functions are found in *wctype.h*.

iswalnum	int	iswalnum	(wint_t a_char);
iswalpha	int	iswalpha	(wint_t a_char);
iswcntrl	int	iswcntrl	(wint_t a_char);
iswdigit	int	iswdigit	(wint_t a_char);
iswgraph	int	iswgraph	(wint_t a_char);
iswlower	int	iswlower	(wint_t a_char);
iswprint	int	iswprint	(wint_t a_char);
iswpunct	int	iswpunct	(wint_t a_char);
iswspace	int	iswspace	(wint_t a_char);
iswupper	int	iswupper	(wint_t a_char);
iswxdigit	int	iswxdigit	(wint_t a_char);
towlower	int	towlower	(wint_t a_char);
towupper	int	towupper	(wint_t a_char);

F.3 Math Library

The following functions are found in *math.h*.

acos	double	acos	(double number);
acosf	float	acosf	(float number);
acosl	long double	acosl	(long double number);
asin	double	asin	(double number);
asinf	float	asinf	(float number);
asinl	long double	asinl	(long double number);
atan	double	atan	(double number);
atanf	float	atanf	(float number);
atanl	long double	atanl	(long double number);
atan2	double	atan2	(double number1, double number2);
atan2f	float	atan2f	(float number1, float number2);
atan2l	long double	atan2l	(long double number1, long double number2);
ceil	double	ceil	(double number);
ceilf	float	ceilf	(float number);
ceill	long double	ceill	(long double number);
cos	double	cos	(double number);
cosf	float	cosf	(float number);
cosl	long double	cosl	(long double number);
cosh	double	cosh	(double number);
coshf	float	coshf	(float number);
coshl	long double	coshl	(long double number);
exp	double	exp	(double number);

expf	float	expf	(float number);
expl	long double	expl	(long double number);
expm1	double	expm1	(double number);
expm1f	float	expm1f	(float number);
expm1l	long double	expm1l	(long double number);
exp2	double	exp2	(double number);
exp2f	float	exp2f	(float number);
exp2l	long double	exp2l	(long double number);
fabs	double	fabs	(double number);
fabsf	float	fabsf	(float number);
fabsl	long double	fabsl	(long double number);
floor	double	floor	(double number);
floorf	float	floorf	(float number);
floorl	long double	floorl	(long double number);
fmod	double	fmod	(double number1, double number2);
fmodf	float	fmodf	(float number1, float number2);
fmodl	long double	fmodl	(long double number1, long double number2);
frexp	double	frexp	(double number, int* exponent);
frexpf	float	frexpf	(float number, int* exponent);
frexpl	long double	frexpl	(long double number, int* exponent);
ilogb	double	ilogb	(double number);
ilogbf	float	ilogbf	(float number);
ilogbl	long double	ilogbl	(long double number);
ldexp	double	ldexp	(double number, int power);
ldexpf	float	ldexpf	(float number, int power);
ldexpl	long double	ldexpl	(long double number, int power);
lint	double	lint	(double number);
lintf	float	lintf	(float number);
lintl	long double	lintl	(long double number);
lrint	long	lrint	(double number);
lrintf	long	lrintf	(float number);
lrintl	long	lrintl	(long double number);
llrint	long long	llrint	(double number);
llrintf	long long	llrintf	(float number);
llrintl	long long	llrintl	(long double number);
log	double	log	(double number);
logf	float	logf	(float number);
logl	long double	logl	(long double number);
log2	double	log2	(double number);
log2f	float	log2f	(float number);
log2l	long double	log2l	(long double number);
log10	double	log10	(double number);
log10f	float	log10f	(float number);
log10l	long double	log10l	(long double number);
lround	long	lround	(double number);
lroundf	long	lroundf	(float number);
lroundl	long	lroundl	(long double number);
llround	long long	llround	(double number);
llroundf	long long	llroundf	(float number);
llroundl	long long	llroundl	(long double number);
modf	double	modf	(double number, double* integral);

modff	float	modff	(float number, float* integral);
modfl	long double	modfl	(long double number, long double* integral);
nearbyint	double	nearbyint	(double number);
nearbyintf	float	nearbyint	(float number);
nearbyintl	long double	nearbyintl	(long double number);
pow	double	pow	(double base, double power);
powf	float	powf	(float base, float power);
powl	long double	powl	(long double base, long double power);
remainder	double	remainder	(double number1, double number2);
remainderf	float	remainderf	(float number1, float number2);
remainderl	long double	remainderl	(long double number1, long double number2);
remquo	double	remquo	(double number1, double quotient);
remquof	float	remquof	(float number1, float quotient);
remquol	long double	remquol	(long double number1, long double quotient);
rint	double	rint	(double number);
rintf	float	rintf	(float number);
rintl	long double	rintl	(long double number);
round	double	round	(double number);
roundf	float	roundf	(float number);
roundl	long double	roundl	(long double number);
scalebn	double	scalebn	(double number, int factor);
scalebnf	float	scalebnf	(float number, int factor);
scalebnl	long double	scalebnl	(long double number, int factor);
scalebln	double	scalebln	(double number, long factor);
scaleblnf	float	scalebnlf	(float number, long factor);
scaleblnl	long double	scalebnll	(long double number, long factor);
sin	double	sin	(double number);
sinf	float	sinf	(float number);
sinl	long double	sinl	(long double number);
sinh	double	sinh	(double number);
sinhf	float	sinhf	(float number);
sinhl	long double	sinhl	(long double number);
sqrt	double	sqrt	(double number);
sqrtf	float	sqrtf	(float number);
sqrtl	long double	sqrtl	(long double number);
tan	double	tan	(double number);
tanf	float	tanf	(float number);
tanl	long double	tanl	(long double number);
tanh	double	tanh	(double number);
tanhf	float	tanhf	(float number);
tanhl	long double	tanhl	(long double number);
trunc	double	trunc	(double number);
truncf	float	truncf	(float number);
truncl	long double	truncl	(long double number);

F.4 Standard I/O Library

We have divided the system input/output library (stdio.h) by the type of data being read or written.

General I/O

General input/output contains functions that apply to all files.

clearerr	void	clearerr	(FILE* sp);
fclose	int	fclose	(FILE* sp);
feof	int	feof	(FILE* sp);
ferror	int	ferror	(FILE* sp);
fopen	FILE*	fopen	(const char* extn_name, const char* file_mode);
freopen	FILE*	freopen	(const char* extn_name, const char* file_mode, FILE* stream);

Formatted I/O

Convert text data to/from internal memory formats.

fprintf	int	fprintf	(FILE* fileOut, const char* format_string, ...);
fscanf	int	fscanf	(FILE* fileIn, const char* format_string, ...);
printf	int	printf	(const char* format_string, ...);
scanf	int	scanf	(const char* format_string, ...);
snprintf	int	snprintf	(char* to_loc, size_t n, const char* format_string, ...);
sprintf	int	sprintf	(char* to_loc, const char* format_string, ...);
sscanf	int	sscanf	(const char* from_loc, const char* format_string, ...);

The following wide-character functions also require the *wchar.h* library file.

fwprintf	int	fwprintf	(FILE* fileOut, const wchar_t* format_string, ...);
fwscanf	int	fwscanf	(FILE* fileIn, const wchar_t* format_string, ...);
swprintf	int	swprintf	(wchar_t* to_loc, const wchar_t* format_string, ...);
swscanf	int	swscanf	(const wchar_t* from_loc, const char* format_string, ...);
wprintf	int	wprintf	(const wchar_t* format_string, ...);
wscanf	int	wscanf	(const wchar_t* format_string, ...);

Character I/O

Read and write one character at a time.

fgetc	int	fgetc	(FILE* sp);
fputc	int	fputc	(int char_out, FILE* sp);
getc	int	getc	(FILE* sp);
getchar	int	getchar	(void);
putc	int	putc	(int char_out, FILE* sp);
putchar	int	putchar	(int char_out);
ungetc	int	ungetc	(int char_out, FILE* sp);

The following wide-character functions also require the *wchar.h* library file.

fgetwc	wint_t	fgetwc	(FILE* sp);
fputwc	wint_t	fputwc	(wchar_t char_out, FILE* sp);
getwc	wint_t	getwc	(FILE* sp);
getwchar	wint_t	getwchar	(void);
putwc	wint_t	putwc	(wchar_t char_out, FILE* sp);
putwchar	wint_t	putwchar	(wchar_t char_out);
ungewtc	wint_t	ungetwc	(int char_out, FILE* sp);

File I/O

These functions work with binary files.

fread	size_t	fread	(void* in_area, size_t size, size_t count, FILE* sp);
fseek	int	fseek	(FILE* sp, long offset, int from_loc);
ftell	long	ftell	(FILE* sp);
fwrite	size_t	fwrite	(const void* out_data, size_t size, size_t count, FILE* sp);
rewind	void	rewind	(FILE* sp);

String I/O

Read and write strings.

fgets	char*	fgets	(char* string, int size, FILE* sp);
fputs	int	fputs	(const char* string, FILE* sp);
gets	char*	gets	(char* string);
puts	int	puts	(const char* string);

The following wide-character functions also require the *wchar.h* library file.

fgetws	wchar_t*	fgetws	(wchar_t* string, int size, FILE* sp);
fputws	wchar_t*	fputws	(const wchar_t* string, FILE* sp);

System File Control

System commands that create and delete files on the disk.

remove	int	remove	(const char* file_name);
rename	int	rename	(const char* old_name, const char* new_name);
tmpfile	FILE*	tmpfile	(void);
tmpnam	char*	tmpnam	(char* file_name);

F.5 Standard Library

The following functions are found in *stdlib.h*.

Math Functions

The following math functions are found in *stdlib.h*.

abs	int	abs	(int number);
div	div_t	div	(int numerator, int divisor);
labs	long	labs	(long number);
llabs	long long	llabs	(long number);
ldiv	ldiv_t	ldiv	(long numerator, long divisor);
lldiv	lldiv_t	lldiv	(long long numerator, long long divisor);
rand	int	rand	(void);
srand	void	srand	(unsigned seed);

Memory Functions

The following are memory allocation functions.

calloc	void*	calloc	(size_t num_elements, size_t element_size);
free	void	free	(void*);
malloc	void*	malloc	(size_t num_bytes);
realloc	void*	realloc	(void* stge_ptr, size_t element_size);

Program Control

The following functions control the program flow.

abort	void	abort	(void);
atexit	int	atexit	(void (*) function_name (void));
exit	void	exit	(int exit_code);
_Exit	void	_Exit	(int exit_code);

System Communication

The following function communicates with the operating system.

system	int	system	(const char* system_command);

Conversion Functions

The following functions convert data from one type to another.

atof	double	atof	(const char* string);
atoi	int	atoi	(const char* string);
atol	long	atol	(const char* string);
atoll	long long	atoll	(const char* string);
strtod	double		strtod(const char* str, char** next_str);
strtof	float	strtof	(const char* str, char** next_str);
strtol	long	strtol	(const char* str, char** next_str, int base);
strtold	long double	strtold	(const char* str, char** next_str);
strtoll	long long	strtoll	(const char* str, char** next_str, int base);
strtoul	unsigned long	strtoul	(const char* str, char** next_str, int base);
strtoull	unsigned long long	strtoull	(const char* str, char** next_str, int base);

F.6 String Library

The following functions are found in *string.h*.

Memory Functions

The following functions manipulate block data in memory.

memchr	void*	memchr	(const void* mem, int a_char, size_t bytes);
memcmp	int	memcmp	(const void* mem1, const void* mem2, size_t bytes);
memcpy	void*	memcpy	(void* to_mem, const void* fr_mem, size_t bytes);
memmove	void*	memmove	(void* to_mem, const void* fr_mem, size_t bytes);

String Functions

The following functions manipulate strings.

strcat	char*	strcat	(char* to_str, const char* fr_str);
strchr	char*	strchr	(const char* str, int a_char);
strcmp	int	strcmp	(const char* str1, const char* str2);
strcpy	char*	strcpy	(char* to_str, const char* fr_str);
strlen	size_t	strlen	(const char* str);
strncat	char*	strncat	(char* to_str, const char* fr_str, size_t bytes);
strncmp	int	strncmp	(const char* str1, const char* str2, size_t bytes);
strncpy	char*	strncpy	(char* to_str, const char* fr_str, size_t bytes);
strpbrk	char*	strpbrk	(const char* str1, const char* str2);
strcspn	size_t	strcspn	(const char* str1, const char* str2);
strrchr	char*	strrchr	(const char* str, int a_char);
strspn	size_t	wstrspn	(const char* str1, const char* str2);
strstr	char*	strstr	(const char* str1, const char* str2);
strtok	char*	strtok	(char* str1, const char* str2);

The following functions manipulate wide-character strings. They require the *wchar.h* library.

wcscat	wchar_t*	wcscat	(wchar_t* to_str, const wchar_t* fr_str);
wcschr	wchar_t*	wcschr	(const wchar_t* str, wchar_t a_char);
wcscmp	int	wcscmp	(const wchar_t* str1, const wchar_t* str2);
wcscpy	wchar_t*	wcscpy	(wchar_t* to_str, const wchar_t* fr_str);
wcscspn	size_t	wcscspn	(const wchar_t* str1, const wchar_t* str2);
wcslen	size_t	wcslen	(const wchar_t* str);
wcsncat	wchar_t*	strncat	(wchar_t* to_str, const wchar_t* fr_str, size_t bytes);
wcsncmp	int	wstrncmp	(const wchar_t* str1, const wchar_t* str2, size_t bytes);
wcsncpy	wchar_t*	wcsncpy	(wchar_t* to_str, const wchar_t* fr_str, size_t bytes);
wcspbrk	wchar_t*	wcspbrk	(const wchar_t* str1, const wchar_t* str2);
wcsrchr	wchar_t*	wcsrchr	(const wchar_t* str, wchar_t a_char);
wcsspn	size_t	wcsspn	(const wchar_t* str1, const wchar_t* str2);
wcsstr	wchar_t*	wcsstr	(const wchar_t* str1, const wchar_t* str2);
wcstok	wchar_t*	wcstok	(wchar_t* str1, const wchar_t* str2);

F.7 Time Library

The following functions are found in *time.h*.

clock	clock_t	clock	(void);
difftime	double	difftime	(time_t time_start, time_t time_end);
mktime	time_t	mktime	(struct tm* cal_time);
time	time_t	time	(time_t* num_time);
asctime	char*	asctime	(const struct tm* cal_time);
ctime	char*	ctime	(const time_t* num_time);
gmtime	struct tm*	gmtime	(const time_t* num_time);
localtime	struct tm*	localtime	(const time_t* num_time);
strftime	size_t	strftime	(char* str, size_t maxsize, const char* format, const struct tm* timeptr);

Appendix G Preprocessor Commands

The C compiler is made of two functional parts: a preprocessor and a translator. The preprocessor uses programmer-supplied commands to prepare the source program for compilation. The translator converts the C statements into machine code that it places in an object module. Depending on the compiler design, the preprocessor and translator can work together, or the preprocessor can create a separate version of the source program, which is then read by the translator. This is the design shown in Figure G-1.

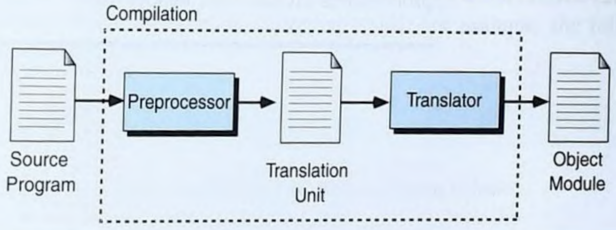


FIGURE G-1 Compiler Components

The preprocessor can be thought of as a smart editor. Like a smart editor, it inserts, includes, excludes, and replaces text based on commands supplied by the programmer. In this case, however, the commands are made a permanent part of the source program.

All **preprocessor commands** start with a pound sign (#). Some of the traditional compilers require the pound sign to be in the first column. ANSI/ISO C specifies that it can be anywhere on the line. Preprocessor commands can be placed anywhere in the source program. To distinguish a preprocessor command and a program command, the preprocessor commands are often called commands.

In this appendix, we first discuss three major tasks of a preprocessor: file inclusion, macro definition, and conditional compilation. We then briefly discuss less common commands such as line, error, and pragma.

G.1 File Inclusion

The first and most common job of a preprocessor is **file inclusion**, the copying of one or more files into programs. The files are usually header files that contain function and data declarations for the program, but they can contain any valid C statement.

The preprocessor command is ***#include***, and it has two different formats. The first format is used to direct the preprocessor to include header files from the system library. In this format, the name of the header file is enclosed in pointed brackets. The second format makes the preprocessor look for the header files in the user-defined directory. In this format, the name of the file pathname is enclosed in double quotes. The two formats are shown below:

```
#include <filename.h>
#include "filepath.h"
```

Figure G-2 shows the situation of the source file before and after the preprocessor as included the header files.

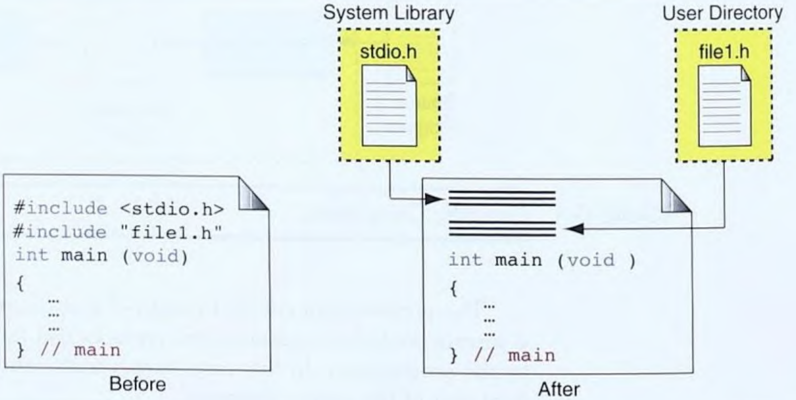


FIGURE G-2 File Inclusion

G.2 Macro Definition

The second preprocessor task is expanding **macro definition**. A macro definition command associates a name with a sequence of tokens. The name is called the *macro name* and the tokens are referred to as the *macro body.* A macro definition has the following form:

```
#define name body
```

The body is the text that is used to specify how the name is replaced in the program before it is translated. Before we discuss different applications of the macro definition, let us clarify two important issues:

1. Macros must be coded on a single line. If the macro is too long to fit on a line, we must use a continuation token. The continuation token is a backslash (\) followed immediately by a newline. If any whitespace is present between the backslash and the newline, then the backslash is not a continuation token, and the code will most likely generate an error. An example of a macro continuation is shown below:

```
#define PREND \
        printf ("Normal end of program PA5-01./n");
```

2. We need to be careful in coding the macro body. Whenever a macro call (name of the macro in the program) is encountered, the preprocessor replaces the call with the macro body. If the body is not created carefully, it may create an error or undesired result. For example, the following macro definition:

```
#define ANS = 0
```

creates a compile error when it is used as shown below:

```
...
num = ANS;
...
```

After preprocessing, the result would be

```
num == 0;                          // We needed  num = 0;
```

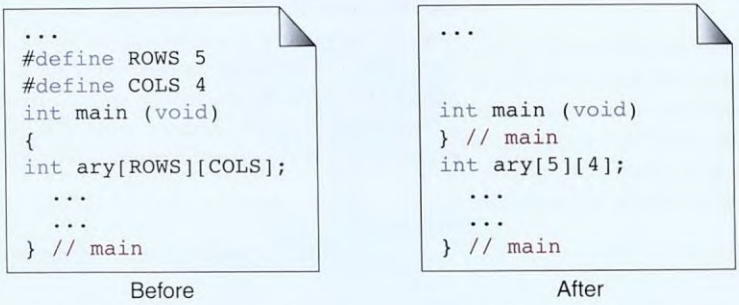
which is not what we wanted.

Coding Defined Constants

The simplest application of a macro is to define a constant. As we saw in Chapter 2, this is one of the ways we define a constant. The following shows an example of a define command for constant definition. The name is SIZE and the body is 9. Whenever in the program SIZE is encountered, it is replaced with 9.

```
#define SIZE 9
```

Figure G-3 shows how we use constant definitions. As the figure shows, the replacement can happen anywhere in the code, including the declaration section.



```
...
#define ROWS 5
#define COLS 4
int main (void)
{
int ary[ROWS][COLS];
  ...
  ...
} // main
```

Before

```
...



int main (void)
} // main
int ary[5][4];
  ...
  ...
} // main
```

After

FIGURE G-3 Macro Definition to Define Constants

The body of the macro definition can be any constant value including integer, real, character, or string. However, character constants must be enclosed in single quotes and string constants in double quotes.

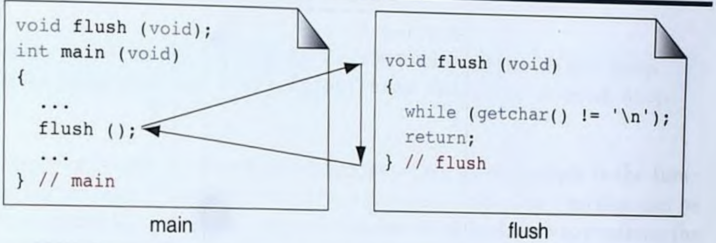
A macro definition can simulate a constant definition.

Macros That Simulate Functions

The preprocessor's macro facility is very powerful. It can even be used to simulate functions. In this section we discuss using macros in place of functions, first for functions with no parameters and then for functions with parameters.

Macros to Simulate Functions without Parameters

When we simulate a function with a macro, the macro definition replaces the function definition. The macro name serves as the header and the macro body serves as the function body. The name of the macro is used in the program to replace the function call. Figure G-4 shows how we can write a function to flush the standard input buffer.

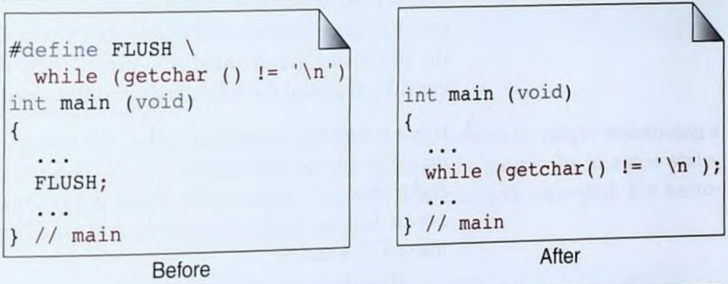


```
void flush (void);
int main (void)
{
   ...
   flush ();
   ...
} // main
```

```
void flush (void)
{
  while (getchar() != '\n');
  return;
} // flush
```

FIGURE G-4 A Function for Flushing the Standard Input Buffer

While functions are called when the program is executed, macro definitions are inserted into the code during the preprocessing. The preprocessor replaces the macro name in the program with the macro body. The macro definition is simpler than the function, however, because we do not have to write the function declaration, the function header, or the return statement. It is also more efficient because the overhead associated with a function call is not required; the substituted code simply becomes statements in the program. Figure G-5 shows how we can use a macro definition to simulate the function in Figure G-4.



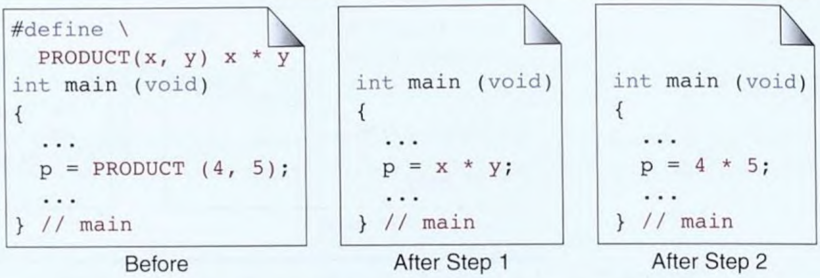
```
#define FLUSH \
  while (getchar () != '\n')
int main (void)
{
  ...
  FLUSH;
  ...
} // main
```

```
int main (void)
{
  ...
  while (getchar() != '\n');
  ...
} // main
```

FIGURE G-5 Macro to Simulate a Function Call Without Parameters

Macros to Simulate Functions with Parameters

We can also use macro definitions to simulate a simple function with parameters. In this case, however, the replacement of the macro is done in two steps, as shown in Figure G-6.



```
#define \
  PRODUCT(x, y) x * y
int main (void)
{
  ...
  p = PRODUCT (4, 5);
  ...
} // main
```

Before

```
int main (void)
{
  ...
  p = x * y;
  ...
} // main
```

After Step 1

```
int main (void)
{
  ...
  p = 4 * 5;
  ...
} // main
```

After Step 2

FIGURE G-6 A Macro to Simulate a Function Call with Parameters

1. The body of the macro replaces the macro call with the same actual parameters used in the macro definition.
2. The actual parameters are replaced with formal parameters.

One of the advantages of using macros instead of function is type independence. We can define a PRODUCT macro that multiplies any two pairs of data type. We do not have to write separate macros, one for each pair.

We need to give two warnings:

1. To include parameters in the macro, the opening parenthesis must be placed immediately at the end of the macro name; that is, there can be no whitespace between the macro name and the opening parenthesis of the parameter list. If there is a space, then the opening parenthesis is considered part of the token body and the macro is assumed to be simple.
2. It is a strong recommendation that the formal parameters, in the body of macro be placed inside parentheses. The reason is that macro uses text replacement. It replaces the actual parameters with formal parameters. Let us look at the same example, but this time we pass a +1 and b +2 instead of 4 and 5:

   After the first and second step, we have:

   ```
   p = x  *  y;
   p = a + 1  *  b + 2 ;          // We need (a +1) * (b+2)
   ```

   The solution would be to include the formal parameters inside parentheses. In other words, make the parentheses part of body code. Now we have the following:

   ```
   #define PRODUCT(x, y) (x) * (y)
   ```

After the first and second step, we have the correct result:

```
p = (x) * (y);                          // After first step
p = (a + 1) * (b + 2);                  // After second step
```

EXAMPLE G-1 Macros can replace many simple functions. One good example is the function we wrote in Chapter 14 to rotate a number *n* bits. This function can be written as a macro as shown below. Note the use of backslash to continue the definition on the next line.

```
#define ROTATE_LEFT(x , n) \
        ( ( (x) << (n) ) | ( (x) >> (32 -x) ) )
```

EXAMPLE G-2 We often need to use a power of 2 in a program. To calculate 2^x, we can easily use the shift operator in a macro as shown below:

```
#define POWER2(x) 1 << (x)
```

EXAMPLE G-3 The parameter list in the macro definition can be empty. For example, most C implementations use a macro to define the *getchar* function from the *getc* function.

```
#define getchar() getc (stdin)
```

Nested Macros

It is possible to nest macros. C handles nested macros by simply rescanning a line after macro expansion. Therefore, if an expansion results in a new statement with a macro, the second macro will be properly expanded. For example, consider the macros shown below.

```
#define PRODUCT(a, b) (a) * (b)
#define SQUARE(a) PRODUCT(a, a)
```

The expansion of

```
x = SQUARE (5);
```

results in the following expansion:

```
x = PRODUCT(5, 5);
```

which after rescanning becomes

```
x = (5) * (5);
```

Undefining Macros

Once defined, a macro command cannot be redefined. Any attempt to redefine it will result in a compilation error. However, it is possible to redefine a macro by first undefining it, using the *#undef* command and defining it again as shown below:

```
#define SIZE  10
...
#undef  SIZE
#define SIZE 20
```

Predefined Macros

The C language provides several predefined macros. Table G-1 lists some of them. Note that these macros cannot be undefined using the *undef* command.

Command	Meaning
_ _DATE_ _	Provides a string constant in the form "Mmm dd yyyy" containing the date of translation.
_ _FILE_ _	Provides a string constant containing the name of the source file.
_ _LINE_ _	Provides an integer constant containing the current statement number in the source file.
_ _TIME_ _	Provides a string constant in the form "hh:mm:ss" containing the time of the translation.
_ _STDC_ _	Provides an integer constant with value 1 if and only if the compiler confirms with ISO implementation.

TABLE G-1 Predefined Macros

Program G-1 demonstrates these macros.

PROGRAM G-1 Demonstrate Pre-Defined Macros

```
/* Show the use of pre-defined macros
      Written by:
      Date:
```

continued

PROGRAM G-1 Demonstrate Pre-Defined Macros *(continued)*

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

int main (void)
{
// Statements
   printf ("line %d\n",   __LINE__);
   printf ("file %s\n",   __FILE__);
   printf ("date %s\n",   __DATE__);
   printf ("time %s\n",   __TIME__);
   printf ("ISO compliance %d\n",   __STDC__);
   printf ("line %d\n",   __LINE__);

   return 0;
}  // main
```

```
Results:
line 10
file ApG-01.c
date May 23 2005
time 20:36:28
ISO compliance 1
line 15
```

Note that we use the _ _LINE_ _ macros two times to show that it prints the current line number.

Operators Related to Macros

C provides several operators that are directly or indirectly related to macros. We briefly discuss them here.

String Converting Operator (#)

The string converting operator (#) is a macro operation that converts a formal parameter into a string surrounded by quotes. For example, the following macro prints the name of a variable followed by its value. It can be a very helpful macro when debugging a program.

```
#define PRINT_VAL(a) printf (#a " contains: %d\n", (a))
```

When called in a program, as shown in the following example

```
PRINT_VAL (amt);
```

the preprocessor expands the macro to

```
printf ("amt" "contains: %d\n", amt);
```

Recall that the preprocessor automatically concatenates two string literals into one string. After the automatic concatenation, the statement becomes

```
printf ("amt contains: %d\n", amt);
```

Merge Operator (##)

Occasionally, it may be necessary to write macros that generate new tokens. With the merge command operator, two tokens are combined. For example, imagine we want to create `A1`, `B3`, and `Z8` in our program. This can be done easily with the following macro definition:

```
#define FORM(T, N) T##N
```

Now if we use the following code:

```
int   FORM (A, 1) = 1;
float FORM (B, 3) = 1.1;
char  FORM (Z, 8) = 'A';
```

we get

```
int   A1 = 1;
float B3 = 1.1;
char  Z8 = 'A';
```

The *defined* Operator

The *defined* operator can be used only in a conditional compilation (see Section G.3); it cannot be used in macros. The value of *defined* `(macro-name)` is 0 if the name is not defined and 1 if it is defined. For example, after

```
#define PI 3.14
```

the value of `defined (PI)` is 1 and the value of `!defined (PI)` is 0.

G.3 Conditional Compilation

The third use of the preprocessor commands is conditional compilation. **Conditional compilation** allows us to control the compilation process by including or excluding statements.

Two-Way Commands

The two-way command tells the preprocessor to select between two choices. This is similar to selection, which we studied in Chapter 5. The idea is shown in Figure G-7.

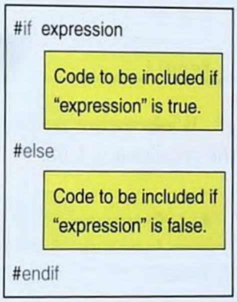

#if expression
Code to be included if "expression" is true.
#else
Code to be included if "expression" is false.
#endif

FIGURE G-7 Two-Way Command

The expression is a constant value that evaluates to zero or non-zero. If the value is non-zero, it is interpreted as *true* and the code after `#if` is included. If it is zero, it is interpreted as *false* and the code after `#else` is executed.

There are two differences between this format and the one used in Chapter 5. First, the expression does not to have to be included in parentheses, although it helps readability when we use parentheses. Second, the codes can not be included in a compound statement (braces). The commands `#else` and `#endif` serve as delimiters.

The `#if` part or the else part can be empty. However, in these cases, the alternative formats are normally used as shown in Figure G-8.

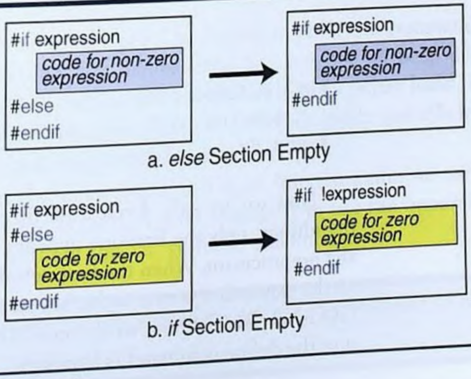


FIGURE G-8 Alternative Commands

EXAMPLE G-4 Let us see how a conditional command can help us in program development. Imagine we need to include a very large file in our program. However, we want to test the syntax of our program gradually, first without the included file and then with it. One way we can do this is to include the file in our program using conditional commands as shown below:

```
#if 0
    #include "large.h"
#endif
```

When we are ready to test the syntax for the entire program, we change the condition to true.

```
#if 1
    #include "large.h"
#endif
```

As we see in a later example, there are easier and better ways to do this.

EXAMPLE G-5 Imagine we need to include three files in our program: `file1.h`, `file2.h`, `file3.h`. These files contain generalized code that are used in projects. Each file must therefore be able to stand alone. When combined, the result may be multiple inclusion of common macros. If the macros are duplicated, we get a compilation error because the preprocessor does not allow duplication of macros.

To prevent this error we have a project standard: whenever we need to define a macro, we define it in a conditional command using the *defined* operator we discussed in "The *defined* Operator" in Section G.2. So, instead of

```
#define GO 0
```

we use

```
#if !defined (GO)
    #define GO 0
# endif
```

Now we are safe. Even if all three files have used the same conditional definitions, only the first one appears in the program; the rest are skipped by the preprocessor. When the first definition is encountered, GO is not defined, so the expression is true and it becomes defined. In the second and third files, GO is already defined, so the conditional compilation expressions are false, and the define command is bypassed.

To be safe, always use macro definitions in a conditional command.

Two Abbreviations

You may have guessed that the previous commands are used often in source files. The C language provides two additional macro commands as shown in Figure G-9.

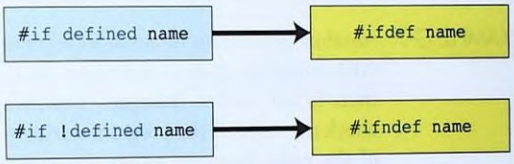


FIGURE G-9 Two Abbreviations

EXAMPLE G-6 When we decompose a large project into different programs, it is common to create a library file that contains all of the application data types and structures. Each program in the project then includes the library file, ensuring that everyone has exactly the same data names and descriptions.

This design creates a problem, however, when multiple files are compiled together. Each file has the same library included for the data, which results in compile errors with the duplicate names. The solution is to enclose the data library file in a defined command as shown in the next example.

```
#ifndef PROG_LIB
    #define PROG_LIB
    #include "proj_lib.h"
#endif
```

Study this code carefully. The nested preprocessor commands can be executed only if a constant named `PROG_LIB` has not yet been defined. In this case, we define it and include the library. On the other hand, if it has been defined, the statements are skipped. This code easily and efficiently prevents the library from being included more than once.

Note that the name `PROG_LIB` is an internal name we use to show the include command is used. A good strategy is to use the uppercase version of the filename, without the ".h" suffix, as a defined name.

To prevent duplicate inclusions, always use file inclusion in a conditional command.

In most implementations, system header files such as *stdio.h* use the above strategy to prevent multiple inclusions of that file in a source program. For example, an implementation may use

```
# ifndef STDIO
     # define STDIO
     Rest of the code
# endif
```

EXAMPLE G-7 Another application of conditional compilation is found in debugging. Most GUI compliers have a debugger that can be used by the programmer. If there is not such a debugger, however, we can insert *printf* statements to check the value of the program during testing. If we just include them as part of the code, however, when the program is ready for production we must remove them. Later, when we need to change the program, we must re-insert them as needed.

A simpler solution is to insert them conditionally. We enclose the *printf* statement inside the #if...#endif commands to print the debug statements only when we run in the "debug" mode. For example, the following program prints the value of x only when DEBUG is true. After testing is done, we can just set the define command to false and the debug statements are not included in the code. This debug concept is demonstrated in Program G-2.

PROGRAM G-2 Demonstrate Conditional Debugging

```
/* Use of conditional compilation for debugging
      Written by:
      Date:
*/
// Macro to print integer values
#define PRINT_VAL(a) \
   printf ("At line %d--", __LINE__); \
   printf (#a " contains: %d\n", (a))

#define DEBUG 1

#include <stdio.h>

int main (void)
{
// Declarations
   int x;
// Statements
```

continued

PROGRAM G-2 Demonstrate Conditional Debugging *(continued)*

```
19   x = 1023;
20
21   #if DEBUG
22       PRINT_VAL (x);
23   #endif
24
25   // Later in program
26   for (int i = 0; i < 2; i++)
27      {
28      x = x * x;                                 // Square x
29      PRINT_VAL (i); PRINT_VAL(x);
30      } // for
31    return 0;
32   } // main
```

```
Result during testing:
At line 22--x contains: 1023
At line 29--i contains: 0
At line 29--x contains: 1046529
At line 29--i contains: 1
At line 29--x contains: 6287361
```

Program G-2 Analysis

Study this program carefully; you will find it useful when you write production programs. First, study the macro definition in lines 6–8. This macro uses two of the techniques we discussed earlier in the chapter. We begin by displaying the source file line number so that we know where we are within the program. Because we have multiple debugging displays, we need to uniquely identify each one. Then we use the string conversion operator (see Section G.2) to display the variable identifier and its contents.

In the body of the program we display the values of the variables x and `i`. In the debugging of a large program, we would use more displays to track the execution of the program.

Multi-Way Commands

Conditional commands can also be multi-way; selecting one of the choices among several. We saw in Chapter 5 that multi-way selection is possible with the use of else-if construct. In the preprocessor, we use the *#elif* command. Figure G-10 shows the format of multi-way selection in the preprocessor.

For example, when we write a software application that must run at multiple locations, we need to define unique code for each location. Each installation needs unique report headings and perhaps other unique code. One way to do this is shown in Program G-3.

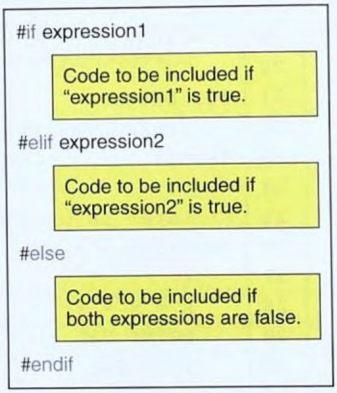

#if expression1
Code to be included if "expression1" is true.
#elif expression2
Code to be included if "expression2" is true.
#else
Code to be included if both expressions are false.
#endif

FIGURE G-10 Multi-Way Commands

ROGRAM G-3 Conditional Multi-Way Selection

```
#define Denver  0
#define Phoenix 0
#define SanJose 1
#define Seattle 0
...
#if (Denver)
   #include "Denver.h"
#elif (Phoenix)
   #include "Phoenix.h"
#elif (SanJose)
   #include "SanJose.h"
#else
   #include "Seattle.h"
#endif
```

m G-3 Analysis First, we create a unique header file for each installation. Then, using conditional commands, we select which include file to use based on defined flags for each site. In this example, we have selected San Jose, so only its code is included in the program.

Summary of Conditional Commands

Table G-2 shows the summary of commands used in this section.

Command	Meaning
#if *expression*	When expression is true, the code that follows is included for compilation.
#endif	Terminates the conditional command.
#else	Specifies alternate code in two-way decision.
#elif	Specifies alternate code in multi-way decision.
#ifdef *name*	*Abbreviation* for #if *defined name.*
#ifndef *name*	*Abbreviation* for #if *!defined name.*

TABLE G-2 Conditional Compilation Commands

G.4 Other Commands

The preprocessor uses some other commands that we briefly discuss here.

Line Command

The line command, which is used in two formats, can set the line number and the filename for a program. For example,

```
#line 100
```

sets the next line of the program to 100. The line number does not show in the source code, but it can be checked by the predefined command `__LINE__`. The second format sets the line number and defines a name for the program. For example,

```
#line 100 "MyProgram.c"
```

sets the next line number to 100 and creates a name for the program that can be checked by the predefined macro call `__FILE__`.

Program G-4 demonstrates the concept.

PROGRAM G-4 Demonstrate Line Command

```
/* Demonstrate the use of line command
      Written by:
            Date:
*/
#line 100 "myprogram.c"
#include <stdio.h>

int main (void)
{

// Statements
   printf ("line %d\n",   __LINE__);
   printf ("file %s\n",   __FILE__);
   printf ("line %d\n",   __LINE__);
   return 0;
} // main
```

```
Results:
line 106
file myprogram.c
line 108
```

Error Command

The error command is of the form

```
#error message
```

It is used to print the message detected by the preprocessor. For example, Program G-5 demonstrates code that verifies that a program needs to define both TRUE and FALSE or neither of them and prints an appropriate message when an error is detected. Note that you may get a slightly different message format depending on your compiler.

PROGRAM G-5 Demonstrate Error Command

```
/* Show the use of error command
      Written by:
      Date:
*/
#define TRUE 1

#if defined (TRUE) && !defined (FALSE)
   #error You need to defined FALSE too.
```

continued

PROGRAM G-5 Demonstrate Error Command *(continued)*

```
#elif defined (FALSE) && !defined (TRUE)
   #error You need to defined TRUE too.
#endif

#include <stdio.h>
int main (void)
{
// Statements
   printf ("Just a test\n");
   return 0;
}  // main
```

```
Results:
Error    : preprocessor #error command
ApG-04.c line 8   #error You need to defined FALSE too.
```

When this program is compiled, we get

```
#error You need to defined FALSE too.
```

Pragma Command

The pragma command

```
#pragma tokens
```

causes the compiler to perform implementation-defined actions. We do not discuss this command in detail because it is used only in advanced environments.

Null Command

It is interesting to mention that C language allows a null command. A command of the type

```
#
```

is considered a null command and does not generate a compilation error.

Appendix H
Command-Line Arguments

In all the programs we have written, *main* has been coded with no parameters. But *main* is a function, and as a function, it may have parameters. When *main* has parameters, they are known as **command-line arguments.**

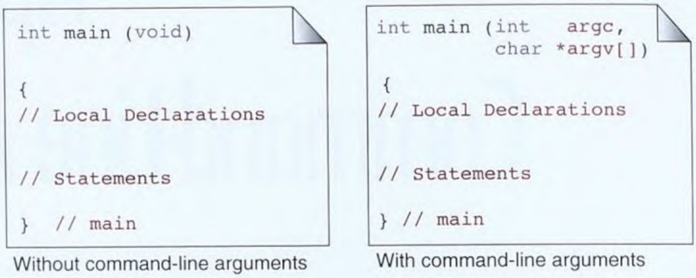
Command-line arguments are parameters to *main* when the program starts. They allow the user to specify additional information when the program is invoked. For instance, if you write a program to append two files, rather than specify the names of the files as constants in the code, the user could supply them when the program starts. Thus, a UNIX user in the Korn Shell might execute the program with the command line shown below.

```
$appendFiles file1 file2
```

H.1 Defining Command-Line Arguments

As the programmer, you design the parameter lists for functions you write. When you use system functions, such as *rand,* you follow the parameter design set up by the language specification. Command-line arguments are a little like both. As the programmer, you have control over the names of the parameters, but their type and format are predefined for the language.

The function *main* can be defined either with no argument (*void*) or with two arguments: one an integer and the other an array of pointers to *char* (strings) that represent user-determined values to be passed to *main.* The number of elements in the array is stored in the first argument. The pointers to the user values are stored in the array. The two different formats are shown in Figure H-1.



```
int main (void)

{
// Local Declarations


// Statements

}  // main
```

Without command-line arguments

```
int main (int   argc,
          char *argv[])

{
// Local Declarations


// Statements

} // main
```

With command-line arguments

FIGURE H-1 Arguments to *main*

Although the names of the arguments are your choice, traditionally they are called ***argc*** (argument count) and ***argv*** (argument vector). This data structure design, with six string pointers in the vector, is shown in Figure H-2.

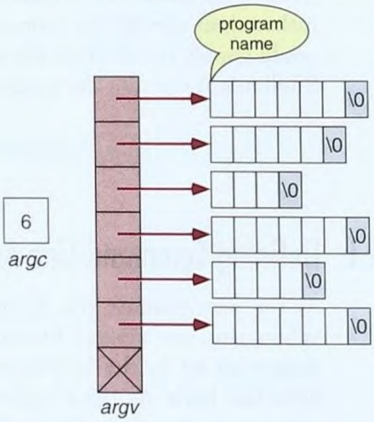


FIGURE H-2 *argc* and *argv* format

The first argument defines the number of elements in the array identified in the second argument. The value for this argument is not entered using the keyboard; the system determines it from the arguments the user types.

The value of *argc* is determined from the user-typed values for *argv's*.

H.2 Using Command-Line Arguments

The *argc* array has several elements. The first element points to the name of the program (its filename). It is provided automatically by the program. The last element contains NULL and may be used to identify the end of the list. The rest of the elements contain pointers to the user-entered string values.

To fully demonstrate how command-line arguments work, let's write a small nonsense program. It does nothing but exercise the command-line arguments. The code is shown in Program H-1.

PROGRAM H-1 Display Command-Line Arguments

```
/* Demonstrate the use of command-line arguments.
      Written by:
      Date:
*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main (int argc, char* argv[])
{
// Statements
   printf ("The number of arguments: %d\n", argc);
   printf ("The name of the program: %s\n", argv[0]);

   for (int i = 1; i < argc; i++)
      printf ("User Value No. %d: %s\n", i, argv[i]);

   return 0;
} // main
```

Now that we've written the program, let's run it with several different arguments. First, let's run it with no user arguments. Even when the user doesn't supply values, the program name is still supplied by the system. (For all of these runs, we assume a Windows-run line environment.)

```
C:>cmdline
The number of arguments: 1
The name of the program: CMDLINE
```

For the second run, let's add hello to the run command.

```
C:>cmdline hello
The number of arguments: 2
The name of the program: CMDLINE
User Value No. 1: hello
```

To make the exercise more interesting, let's run the program with a phrase on the command line.

```
C:>cmdline Now is the time
The number of arguments: 5
The name of the program: CMDLINE
User Value No. 1: Now
User Value No. 2: is
User Value No. 3: the
User Value No. 4: time
```

But what if our intent is to read a phrase? C looks at the user values as strings. This means that the spaces between the words separate each word into a different element. If we want an element to contain more than one word, we enclose it in quotes just like a string in code.

```
C:>cmdline "To err is human" Pope
The number of arguments: 3
The name of the program: CMDLINE
User Value No. 1: To err is human
User Value No. 2: Pope
```

The user elements cannot be numbers. If you enter a number, it is taken as a string. However, the string can be converted to a number using a standard function such as *strtod*.

Appendix I

Pointers to *void* and to Functions

In this appendix we discuss pointer to *void* and pointer to function.

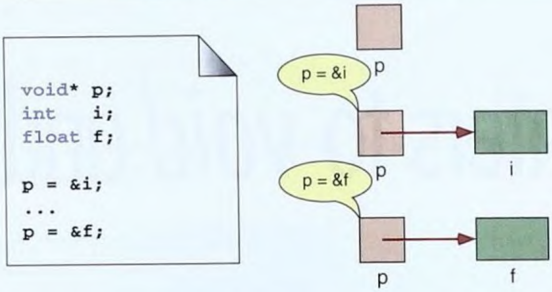
I.1 Pointer to *void*

Because C is strongly typed, operations such as assign and compare must use compatible types or be cast to compatible types. The one exception is the **pointer to *void***, which can be assigned without a cast. In other words, a pointer to *void* is a generic pointer that can be used to represent any data type during compilation or run time. Figure I-1 shows the idea of a pointer to *void*. Note that a pointer to *void* is not a null pointer; it is pointing to a generic data type (*void*).

FIGURE I-1 Pointer to *void*

EXAMPLE I-1 Let us write a simple program to demonstrate the pointer to *void* concept. It contains three variables: an integer, a floating-point number, and a *void* pointer. At different times in the program the pointer can be set to the address of the integer value or of the floating-point value. Figure I-2 shows the situation.

Program I-1 uses a pointer to *void* that we can use to print either an integer or a real number.



```
void* p;
int   i;
float f;

p = &i;
...
p = &f;
```

FIGURE I-2 Pointers for Program I-1

PROGRAM I-1 Demonstrate Pointer to *void*

```
/* Demonstrate pointer to void.
      Written by:
      Date:

*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   void* p;
   int   i = 7;
   float f = 23.5;

// Statements
   p = &i;
   printf ("i contains: %d\n", *((int*)p) );

   p = &f;
   printf ("f contains: %f\n", *((float*)p));
   return 0;
} // main
```

```
Results:
i contains 7
f contains 23.500000
```

Program I-1 Analysis The program is trivial, but it demonstrates the point. The pointer p is declared as a *void* pointer, but it can accept the address of an integer or floating-point number. However, we must remember a very important point about pointers to *void*: a pointer to *void* cannot be dereferenced unless it is cast. In other words, we cannot use `*p` without casting. That is why we need to cast the pointer in the print function before we use it for printing.

A pointer to *void* cannot be dereferenced unless it is cast.

I.2 Pointer to Function

Functions in a program occupy memory. The name of the function is a pointer constant to its first byte of memory. For example, imagine that we have four functions stored in memory: *main*, `fun`, `pun`, and `sun`. This relationship is shown graphically in Figure I-3. The name of each function is a pointer to its code in memory.

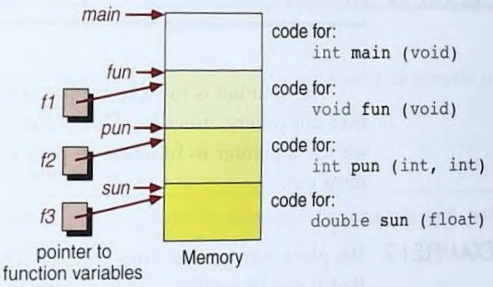

main
fun
f1
pun
f2
sun
f3
pointer to function variables
Memory
code for: int main (void)
code for: void fun (void)
code for: int pun (int, int)
code for: double sun (float)

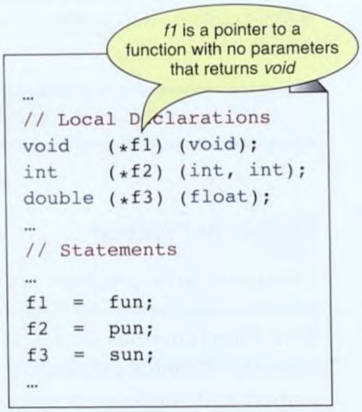
FIGURE I-3 Functions in Memory

Defining Pointers to Functions

Just as with all other pointer types, we can define pointers to function variables and store the address of `fun`, `pun`, and `sun` in them. To declare a **pointer to function**, we code it as if it were a prototype definition, with the function pointer in parentheses. This format is shown in Figure I-4. The parentheses are important: without them C interprets the function return type as a pointer.

Using Pointers to Functions

Now that you've seen how to create and use pointers to functions, let's write a generic function that returns the larger of any two pieces of data. The function uses two pointers to *void* as described in the previous section. While our function needs to determine which of the two values represented by the *void* pointers is larger, it cannot directly compare them because it doesn't know what type casts to use with the *void* pointers. Only the application program knows the data types.



f1 is a pointer to a function with no parameters that returns *void*

```
…
// Local Declarations
void   (*f1) (void);
int    (*f2) (int, int);
double (*f3) (float);
…
// Statements
…
f1  =  fun;
f2  =  pun;
f3  =  sun;
…
```

FIGURE I-4 Pointers to Functions

The solution is to write simple compare functions for each program that uses our generic function. Then, when we call the generic compare function, we use a pointer to function to pass it the specific compare function that it must use.

EXAMPLE I-2 We place our generic function, which we call `larger`, in a header file so that it can be easily used. The program interfaces and pointers are shown in Figure I-5. The code is shown in Program I-2.

PROGRAM I-2 Larger Compare Function

```
/* Generic function to determine the larger of two
   values referenced as void pointers.
      Pre  dataPtr1 and dataPtr2 are pointers to values
             of an unknown type.
           ptrToCmpFun is address of a function that
             knows the data types
      Post data compared and larger value returned
*/
void* larger (void* dataPtr1,    void* dataPtr2,
              int (*ptrToCmpFun)(void*, void*))
{
   if ((*ptrToCmpFun) (dataPtr1, dataPtr2) > 0)
        return dataPtr1;
   else
        return dataPtr2;
}  // larger
```

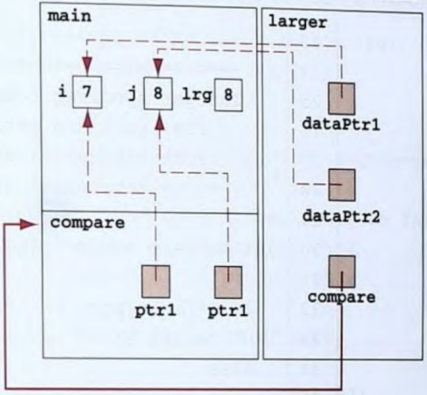


FIGURE I-5 Larger Compare Function

Program I-3 contains an example of how to use our generic compare program and pass it a specific compare function.

PROGRAM I-3 Compare Two Integers

```
/* Demonstrate generic compare functions and pointer to
   function.
      Written by:
      Date:
*/
#include <stdio.h>
#include "ApI-05.h"                              // Header file

int compare (void* ptr1, void* ptr2);

int main (void)
{
// Local Declarations
   int i = 7;
   int j = 8;
   int lrg;

// Statements
   lrg = (*(int*) larger (&i, &j, compare));
   printf("Larger value is: %d\n", lrg);
   return 0;
}  // main
```

continued

PROGRAM I-3 Compare Two Integers *(continued)*

```

/* ==================== compare ====================
   Integer specific compare function.
      Pre  ptr1 and ptr2 are pointers to integer values
      Post returns +1 if ptr1 >= ptr2
           returns -1 if ptr1 <  ptr2
*/
int compare (void* ptr1, void* ptr2)
{
  if (*(int*)ptr1 >=  *(int*)ptr2)
     return 1;
  else
     return -1;
} // compare
```

```
Results:
Larger value is: 8
```

EXAMPLE I-3 Now, let's write a program that compares two real (*float*) numbers. We can use our larger function, but we need to write a new compare function. We repeat Program I-3, changing only the compare function and the data-specific statements in *main*. The result is shown in Program I-4.

PROGRAM I-4 Compare Two Floating-Point Values

```
/* Demonstrate generic compare functions and pointer to
   function.
      Written by:
      Date:
*/
#include <stdio.h>
#include "ApI-02.c"                     // Header file

int compare (void* ptr1, void* ptr2);

int main (void)
{
// Local Declarations
   float i = 73.4;
   float j = 81.7;
   float lrg;

// Statements
   lrg = (*(float*) larger (&i, &j, compare));
```

continued

PROGRAM I-4 Compare Two Floating-Point Values *(continued)*

```
   printf("Larger value is: %.1f\n", lrg);
   return 0;
}  // main

/* ==================== compare ====================
   Float specific compare function.
      Pre  ptr1 and ptr2 are pointers to float values
      Post returns +1 if ptr1 >= ptr2
           returns -1 if ptr1 <  ptr2
*/
int compare (void* ptr1, void* ptr2)
{
  if (*(float*)ptr1 >=  *(float*)ptr2)
     return 1;
  else
     return -1;
}  // compare
```

```
Results:
Larger value is: 81.7
```

Appendix J
Storage Classes and Type Qualifiers

In this appendix, we discuss two attributes of data types, storage classes and type qualifiers. Storage classes specify the scope of objects. Type qualifiers specify processing limitations on objects.

J.1 Storage Classes

We define the storage class of an object using one of four specifiers: *auto*, *register*, *static*, and *extern*. Before we can discuss the storage class specifiers, however, we must describe the environment in which they are used.

Object Storage Attributes

Storage class specifiers control three attributes of an object's storage as shown in Figure J-1: its scope, extent, and linkage.

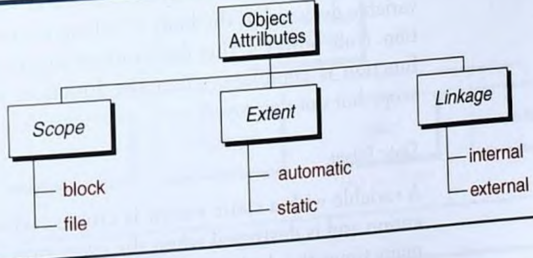


FIGURE J-1 Object Storage Attributes

Scope

Scope defines the visibility of an object; it defines *where* an object can be referenced. In C, an object can have four levels of scope: block, file, function, and function-prototype. In this text, we do not discuss the function and function prototype scopes.

Block (Local) Scope

The body of a function is a block and a compound statement in a loop is a nested block within its function block. When the scope of an object is block, it is visible only in the block in which it is defined. An object with a block scope is sometimes referred to as a local object.

The scope of an object declared anywhere in a block has block scope. For example, a variable declared in the formal parameter list of a function has block scope. A variable declared in the initialization section of a *for* loop also has block scope, but only within the *for* statement.

File (Global) Scope

File scope includes the entire source file for a program, including any files included in it. An object with file scope has visibility through the whole source file in which it is declared. However, objects within block scope are excluded from file scope unless specifically declared to have file scope; in other words, by default block scope hides objects from file scope. File scope generally includes all declarations outside a function and all function headers. An object with file scope is sometimes referred to as a global object.

Extent

The **extent** of an object defines the duration for which the computer allocates memory for it. In C, an object can have automatic, static extent, or dynamic extent. The extent of an object is also known as its storage duration.

Automatic Extent

An object with an automatic extent is created each time its declaration is encountered and is destroyed each time its block is exited. For example, a variable declared in the body of a loop is created and destroyed in each iteration. Note however, that declarations in a function are not destroyed until the function is complete; when the function calls a function, they are out of scope but not destroyed.

Static Extent

A variable with a static extent is created when the program is loaded for execution and is destroyed when the execution stops. This is true no matter how many times the declaration is encountered during the execution.

Dynamic Extent

Dynamic extent is created by the program through the *malloc* and its related library functions. We do not discuss static memory in this text.

Linkage

A large application program may be broken into several modules, with each module potentially written by a different programmer. Each module is a separate source file with its own objects. Different modules may be related when the program is link edited.

We can define two types of **linkage**: internal and external. As we discuss later, the linkage characteristic differs from scope characteristic, although they look the same at the first glance.

Internal Linkage

An object with an internal linkage is declared and visible only in one module. Other modules cannot refer to this object.

External Linkage

An object with an external linkage is declared in one module but is visible in all other modules that declare it with a special keyword, *extern*, as we discuss shortly.

Storage Class Specifiers

We have defined three storage classes—scope, extent, and linkage—with potentially eight different combinations (2 × 2 × 2). If we want to use a specifier for each combination, we need a total of eight different specifiers. However, there are only four **storage class specifiers**, as shown in Figure J-2: *auto*, *register*, *static*, and *extern*. Three factors have contributed to this reduced set.

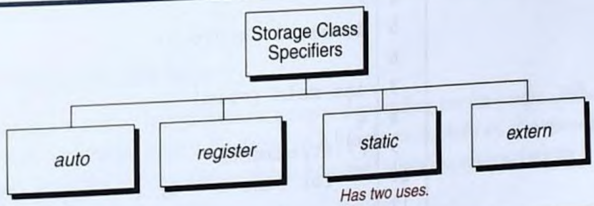


FIGURE J-2 Storage Class Specifiers

1. The scope is not defined, it is understood. The place where an object is declared defines the scope.
2. Not all combinations are logically possible.
3. One specifier, *static*, is used for two different purposes. The scope shows what it defines.

In the rest of the section, we discuss these specifiers only when applied to variables. The application of these specifiers to other objects, such as functions, is similar.

Auto Variables

A variable with an ***auto*** specification has the following storage characteristic:

Scope: block	**Extent: automatic**	**Linkage: internal**

In other words, the variable must be declared in a block. Each time the declaration is encountered, it is re-created. It is visible only in the source file in which it is declared. Most of the variables we have declared so far are *auto* storage class. This type of specifier is so prevalent that C allows us to omit the specifier.

Initialization

An *auto* variable can be initialized where it is defined or left uninitialized. If initialized, it receives the same initialization value each time it is created. If it is not initialized, its value will be undefined every time it is born. When the *auto* variable is not initialized, its value is unpredictable. Program J-1 shows the use of the auto variables.

PROGRAM J-1 Demonstration of *auto* Variables

```
/* Show the use of auto variables
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Statements
   for (int i = 1; i <= 3; i++)
       {
         int x = 1;
         x++;
         printf ("Value of x in iteration %d is: %d\n",
                  i, x);
```

continued

PROGRAM J-1 Demonstration of *auto* Variables *(continued)*

```
      } // for
   return 0;
} // main
```

```
Result:
Value of x in iteration 1 is: 2
Value of x in iteration 2 is: 2
Value of x in iteration 3 is: 2
```

Both i and x are *auto* variables. The declaration of i is encountered only once (*for* loop initialization), but the declaration of x is encountered three times. Each time the program encounters the declaration for x, it is initialized to 1 again.

> **The keyword *auto* does not have to be explicitly coded. Each time a variable of block scope with no specifier is encountered, it defaults to *auto*.**

Register Variables

A **register** storage class is the same as the auto class with only one difference. The declaration includes a recommendation to the compiler to use a central processing unit (CPU) register for the variable instead of a memory location.This is done for efficiency. The time required to access a CPU register is significantly less than the time required to access a memory location. The compiler, however, may ignore the recommendation. Registers are scarce resources, and the compiler often needs all of them for the program to run efficiently.

There is one restriction on the use of a register variable; a register variable address is not available to the user. This means that we can't use the address operator and the indirection operator with a register. It also disallows implicit conversions, such as might occur when using a register with mixed mode arithmetic.

Static Variables with Block Scope

The specifier ***static*** has different usages depending on its scope. When it is used with a variable that is declared in a block, *static* defines the extent of the variable. In other words, the variable has the following characteristic:

Scope: block **Extent: static** **Linkage: internal**

A *static* variable in this context can be referred to only in the block it is defined. The extent, however, is static; the computer allocates storage for this variable only once. The linkage is internal, which means that it is not visible in other modules. Note that the specifier *static* must be explicitly coded in this case; it cannot be omitted.

Initialization

A *static* variable can be initialized where it is defined, or it can be left uninitialized. If initialized, it is initialized only once. If it is not initialized, its value will be initialized to zero. Note however, that it is initialized only once in the execution of the program.

In a static variable with block scope, *static* defines the extent of the variable.

Program J-2 shows the use of static variable in the block scope.

PROGRAM J-2 *Static* Variables in Block Scope

```
/* Show the use of static variable with block scope
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Statements
   for (int i = 1; i <= 3; i++)
   {
    static int x = 1;
    x++;
    printf ("Value of x in iteration %d is: %d\n", i, x);
   } // for
   return 0;
}  // main
```

```
Result:
Value of x in iteration 1 is: 2
Value of x in iteration 2 is: 3
Value of x in iteration 3 is: 4
```

Note that the variable `i` is an *auto* variable. The variable `x`, however, is a *static* variable. It is only initialized once although the declaration is encountered three times. If we do not initialize `x`, it is initialized to 0 because a static value needs an initialization value.

Static Variable with File Scope

When the static specifier is used with a variable that has file scope (global scope) and we want to keep its linkage internal, it is defined with the specifier *static*. Note that the extent is still static. In other words, we have the following characteristic:

Scope: file **Extent: static** **Linkage: internal**

Program J-3 shows the use of the *static* variable with file scope.

PROGRAM J-3 *Static* Variables with File Scope

```
/* Show the use of static variable with file scope
      Written by:
      Date:
*/
#include <stdio.h>

// Function Declaration
void fun (void);

static int x = 1;

int main (void)
{
// Statements
   for (int i = 1; i <= 3; i++)
        x++;
   printf ("Value of x in main is: %d\n", x);
   fun();
   printf ("Value of x in main is: %d\n", x);

   return 0;
}  // main

/* ================== fun ==================
   Increment and print global variable.
      Pre  Global variable x has been defined
      Post x incremented and printed
*/
void fun (void)
{
// Statements
   x++;
   printf ("Value of x in fun is:  %d\n", x);
   return;
}  // fun
```

```
Result:
```

continued

PROGRAM J-3 *Static* Variables with File Scope *(continued)*

```
Value of x in main is: 4
Value of x in fun is:  5
Value of x in main is: 5
```

The declaration of a *static* variable with file scope must be in the global area of the file (outside any function). If there is another declaration with the same identifier in the global area, we get an error message. On the other hand, if the identifier is used with another declaration *inside* a function, the new variable is an *auto* variable or *static* variable with block scope. In other words, in this case, we are defining a new variable that overrides the global variable making it not in the current scope.

External Variables

External variables are used with separate compilations. It is common, on large projects, to decompose the project into many source files. The decomposed source files are compiled separately and linked together to form one unit.

A variable declared with a storage class of ***extern*** has a file scope; the extent is static, but the linkage is external.

Scope: file **Extent: static** **Linkage: external**

An *extern* variable must be *declared* in all source files that reference it, but it can be defined only in one of them. External variables are generally declared at the global level. The keyword *extern* is optional.

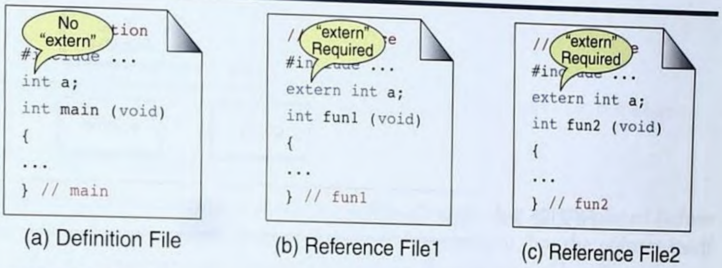
This design can create a problem: if three source files use the *extern* specifier for a variable, which one is the defining file and which are simply declaring the variable? The rules differ depending on whether the variable is initialized or not.

1. If it is not initialized, then the first declaration seen by the linkage editor is considered the definition and all other declarations reference it.

   Figure J-3 contains an example of an external reference used in three programs.

2. If an initializer is used with a global definition, it is the defining declaration.

It is an error if two external variables are initialized. It is also an error to use the same identifier for external variables with incompatible types.



```
           tion
#    clude ...
int a;
int main (void)
{
...
} // main
```

(a) Definition File

```
//          e
#in     ...
extern int a;
int fun1 (void)
{
...
} // fun1
```

(b) Reference File1

```
//           e
#in      ...
extern int a;
int fun2 (void)
{
...
} // fun2
```

(c) Reference File2

FIGURE J-3 Defining and Declaring External References

We recommend that each external definition used in a project be placed in a common "definitions" source file. These definitions should not use the key word *extern* and should have an explicit initializer. Each source file that needs to reference an externally defined variable should use the keyword *extern* and must not use an initializer.

The keyword *extern* must be explicitly used in source files that use an external variable. It is omitted in the source file that defines and initializes the variable.

Summary

We have summarized what we discussed about storage classes in Table J-1.

Class	Scope	Extent	Linkage	Keyword
auto	block	automatic	internal	auto or none
register	block	automatic	internal	register
static (extent)	block	static	internal	static
static (linkage)	file	static	internal	static
extern	file	static	external	extern or none

TABLE J-1 Summary of Storage Classes

J.2 Type Qualifiers

The **type qualifier** (Figure J-4) adds three special attributes to types: *const, volatile, and restrict.*

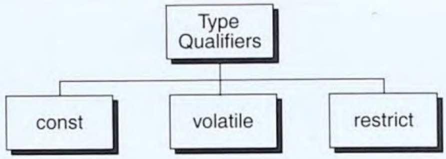


FIGURE J-4 Type Qualifiers

When a storage class, such as *static*, and a type qualifier are both needed, the storage class comes before the type qualifier.

Constants

The keyword for the constant type qualifier is *const*. A constant object is a read-only object; that is, it can only be used as an rvalue. A constant object must be initialized when it is declared because it cannot be changed later. A simple constant is shown below.

```
const double PI = 3.1415926;
...
PI = 3.142 ;                                          // Invalid
```

In addition to the simple types, arrays, structures, and unions can also be defined as constant. In these cases, all their component elements are constants. For example, in the string constant shown below, none of the individual elements can be changed.

```
const char str[] = "Hello";
```

Pointers and Constants

Pointers can also be defined as constants. Depending on how they are coded, however, three different variations can occur.

1. The pointer itself is constant.
2. The object being pointed to is constant.
3. Both the pointer and its object are constants.

Case I: Pointer Constant

When the keyword *const* is associated with the identifier, that is, it is placed after the type and before the identifier, the pointer itself is a constant. This means that its contents cannot be changed after initialized. In other words, the pointer can only point to the object that it was pointing to during initialization; it cannot point to another object. In this case, it is an error to use the pointer as an lvalue.

```
int a;
int b;
int* const ptr = &a;
ptr = &b;                           // Error: ptr is constant
```

Case II: Object Is Constant

When the keyword *const* is associated with the type, that is, it is placed before the type, then the object being referenced is a constant, but the pointer itself is not. So, while we can change the value in the pointer (address), we cannot dereference the operator as an lvalue. In other words, we cannot change the value of the variable through this pointer. Consider the following definitions:

```
int  a = 5;
const int* ptr= &a;
*ptr = 21;                     // Error: pointing to constant
```

Case III: Both Pointer and Object Are Constant

To indicate that both the pointer and the object that it points to are constant, use the keyword *const* twice.

```
int a = 5;
int b = 3;
const int* const p = &a;
*p = 5;                 // Invalid: object of p is a
constant
p  = &b;                // Invalid: p is a constant
```

Volatile

The ***volatile*** qualifier tells the computer that an object value may be changed by entities other than this program. Normally, objects used in a program belong only to the C compiler. When the compiler owns an object, it can store and handle it in any way necessary to optimize the program. As an example, a C compiler may think that it is more efficient to remove an object from RAM memory and put it in a register.

However, sometimes objects must be shared between your program and some other facilities outside the C program, for example, some input/output routines. To tell the compiler that the object is shared, we declare it as type *volatile*. In this case, we are telling the compiler that this object may be referenced or changed by other entities.

The following shows how an integer or a pointer to an integer can be declared *volatile*.

```
volatile  int   x;
volatile  int*  ptr;
```

Restricted

The ***restrict*** qualifier, which is used only with pointers, indicates that the pointer is only the *initial* way to access the dereferenced data. It provides more help to the compiler for optimization. Let us look at the following program segment:

```
int*   ptr;
int    a = 0;
ptr = &a;
...
*ptr += 4;
...
*ptr += 5;
...
```

Here, the compiler cannot replace two statements *ptr += 4 and *ptr +=5 by one statement *ptr += 9 because it does not know if the variable a can be accessed directly or through other pointers.

Now, let us look at the same program fragment using the *restrict* qualifier:

```
restrict int* ptr;
int    a = 0;
ptr = &a;
...
*ptr += 4;
...
*ptr += 5;
...
```

Here, the compiler can replace the two statements by one statement *ptr += 9, because it is sure that variable a cannot be accessed through any other resources.

As an application of the *restrict* qualifier, let us look at two functions in the C library that is used to copy bytes from one location to another:

```
void* memcpy  (      void* restrict destination,
               const void* restrict source, size_t n);

void* memmove (      void* destination,
               const void* source, size_t n);
```

The memory copy function requires that the source and destination area not overlap; the memory move function does not. The restrict qualifier in the first function guarantees that there is no overlap between the source and destination area. The second function does not; the user must be sure that there is no overlap.

Appendix K
Program Development

In Chapter 2, we briefly discussed how a source file is changed to an executable file. In Appendix G we discussed the role of the preprocessor in this process. In this appendix, we show, with more details, how a program, simple or complex, is developed from one single source file, a source file and several user header files, or several source files, into one single executable program.

In the first section, we discuss the process of development, which is common to all platforms. In the second section, we discuss how the process is coded in some common platforms.

K.1 Process

First, let us discuss the process, which creates a single executable program from one or more source files. Although we use four different exhaustive cases, the source program that we have chosen is a simple one. It is a simple program that calculates

$$Y = 2^X$$

where X is entered as input and Y is printed as output. In the first case, we use just the *main* function. In the second, we use a function to do the calculation. In the third, we include the function as the header file. In the fourth, we use two separate source files to show the idea of separate compilations.

First Case: A Simple File

The first case, as shown in Figure K-1, is trivial and seldom seen outside an academic environment. Nevertheless, it illustrates several points.

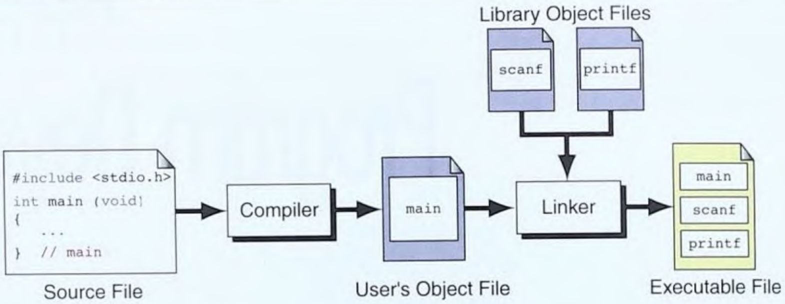


FIGURE K-1 Compilation and Linking of a Single Source File

We have only one source file that contains only one function, *main*. The source file is first compiled to create an **object file**, the binary encoding of the *main* function. The linker adds two library functions, *scanf* and *printf*, to the user's object file to create an **executable file**. The compiler instructs the linker to do so when it encounters the function calls for these two functions. The *scanf* and *printf* functions have already been compiled and stored as object library modules in the standard library *stdio.h*. No compilation is needed for these two functions. Note that the linker can only link (add together) object files. The executable file contains the binary code for these three functions.

Program K-1 shows this situation. We have written only one *main* function to calculate the power of two of any unsigned integer.

PROGRAM K-1 Our Simple Application Program

```
/* Demonstrate a simple compilation
      Written by:
      Date:
*/
#include <stdio.h>
int main (void)
{
// Local Declarations
   unsigned int num;
   unsigned int res;

// Statements
   printf ("Enter the power: ");
   scanf  ("%u", &num);
   res = 1 << num;
   printf ("%u\n", res);
```

continued

PROGRAM K-1 Our Simple Application Program *(continued)*

```
   return 0;
}  // main
```

```
Results:
Enter the power: 14
16384
```

Second Case: Two Functions

In the second case, we still have one source file, but two functions: *main* and *power2*. The system flow is shown in Figure K-2.

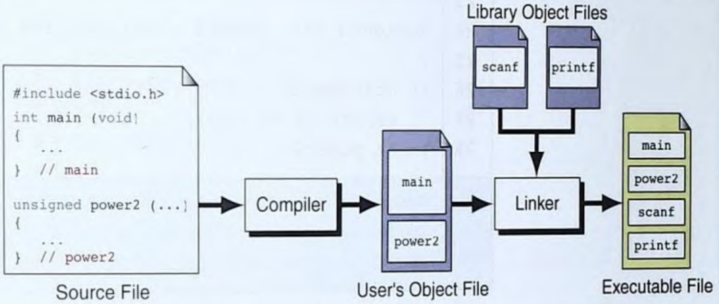


FIGURE K-2 Compilation and Linking of Source File Including a Function

We still have one user's object file and two library module files, but the user's object file now contains binary code for two functions: *main* and *power2*. The linker still links only three object files. The executable file contains the code for four functions.

Program K-2 contains the modified source program with two functions.

PROGRAM K-2 Source Code for *main* and *power2*

```
/* Demonstrate compilation with two functions.
      Written by:
      Date:
*/
#include <stdio.h>

// Function Declaration
unsigned int power2 (unsigned int exp);

int main (void)
```

continued

PROGRAM K-2 Source Code for *main* and *power2 (continued)*

```
{
// Local Declarations
   unsigned int num;
   unsigned int res;

// Statements
   printf ("Enter the power: ");
   scanf  ("%u", &num);
   res = power2 (num);
   printf ("%u\n", res);
   return 0;
}  // main

unsigned int  power2 (unsigned int exp)
{
// Statements
   return (1 << exp);
}  // power2
```

```
Results:
Enter the power: 14
16384
```

Third Case: Two Source Files

The third case is very common in writing structured programs, as you will see in your data structures course. In this trivial situation, we can have one user's source file and one user's header file. The two, however, are associated with each other through the include directive in the source file. Figure K-3 shows the system flow.

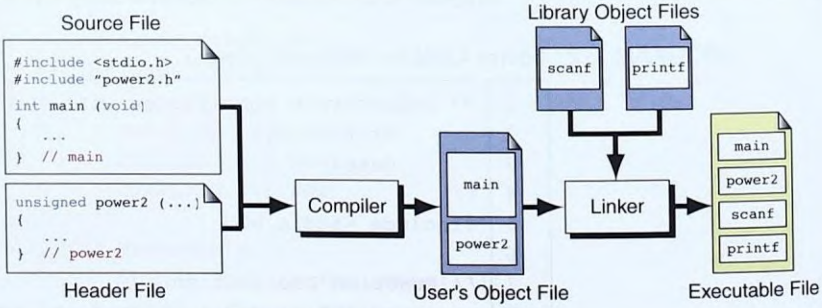


FIGURE K-3 Compilation and Linking of a Source File and a Header File

Still there is one compilation. However, as we saw in Appendix G, the preprocessor needs to act on the include directive and include the user's header file in the source file before translation. The rest of the process is the same as the second case. One of the advantages of this case is that the header file is an independent file that can be reused with any other program. All we need to do to use it is to include its header file. Because the resulting program is identical to the previous version, its results are also the same. Program K-3 shows the source file for *main*.

PROGRAM K-3 Source Code for *main* Only

```
/* Demonstrate compilation with two source files.
      Written by:
      Date:
*/
#include <stdio.h>
#include "power2.h"

int main (void)
{
// Local Declarations
   unsigned int num;
   unsigned int res;
// Statements
   printf ("Enter the power: ");
   scanf  ("%u", &num);
   res = power2 (num);
   printf ("%u\n", res);
   return 0;
 }  // main
```

Program K-4 contains the totally independent `power2.h` file.

PROGRAM K-4 Source File for *power2*

```
/* This is the source file for the power2 function.
      Written by:
      Date:
*/
unsigned int power2 (unsigned int exp)
{
//  Statements
   return (1 << exp);
 }  // power2
```

Case 4: Separate Compilation of Function

This case is more challenging. Not only do we want to make the functions independent, but we also want to compile them separately. Used primarily with generalized software and large projects, it is known as separate compilations.

When a large project is designed, it is decomposed into modules that can be developed and tested separately. It is much easier to write and debug a small module than a large one. After each test file has been separately tested and debugged, the object files can be linked together to form one executable program. Figure K-4 shows the flow for separate compilations.

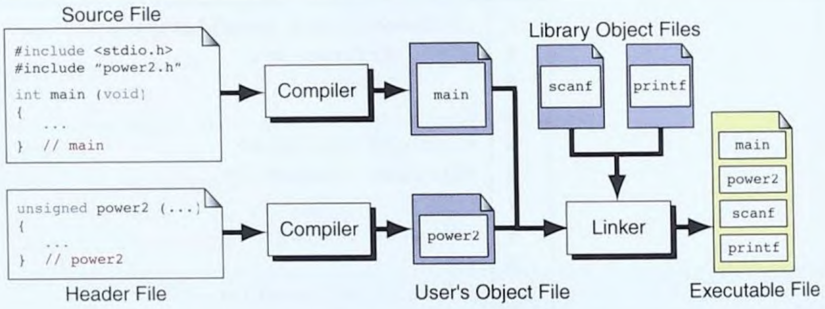


FIGURE K-4 Compilation and Linking of Two Independent Source Files

Although the resulting executable file has the same four binary-coded modules, the process is different. Each source file is compiled separately and two independent object files are made from them. The object files are only related when they are linked together. Once again, because the code is the same, the output is identical.

Program K-5 contains the source file for *main*.

PROGRAM K-5 Separate Compilations—*main*

```
/* Demonstrate the separate compilations—main
      Written by:
      Date:
*/
#include <stdio.h>
int main (void)
{
// Local Declarations
   unsigned int num;
   unsigned int res;

// Statements
```

continued

PROGRAM K-5 Separate Compilations—*main (continued)*

```
   printf ("Enter the power: ");
   scanf  ("%u", &num);
   res = power2 (num);
   printf ("%u\n", res);
   return 0;
}  // main
```

Program K-6 contains the source file for *power2*.

PROGRAM K-6 Separate Compilations—*power2*

```
/* Demonstrate the separate compilations—power2
      Written by:
      Date:
*/
unsigned int power2 (unsigned int exp)
{
// Statements
   return (1 << exp);
}  // power2
```

K.2 Coding

Now that we have discussed the process, let's look at how we code the different steps in the process.

UNIX

UNIX, and its derivative **Linux**, are popular operating systems. They contain several tools with many options. In this appendix, we briefly describe the basic options and explain how to compile and link UNIX programs.

File Types

UNIX uses different extensions to distinguish between different file types, source files, object files, and executable files, as shown in Figure K-5.

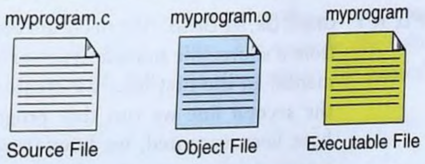


FIGURE K-5 File types in UNIX

In Figure K-5, the source file is `myprogram.c`, the object file is `myprogram.o` and the executable file is `myprogram`. In other words, a source file has the extension "`.c`", an object file has the extension "`.o`", and an executable file has no extension.

In UNIX, a source file in Standard C has the extension *.c*, an object file has the extension *.o*, and an executable file has no extension.

Command and Options

UNIX uses only one command, `c99`, to compile or link different files. The command is used with different options to define compilation only, linking only, or compilation and linking. Figure K-6 shows the general format of the command.

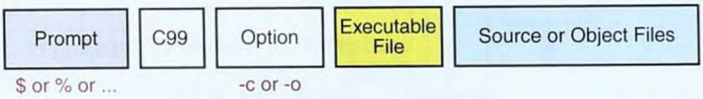


FIGURE K-6 Compilation and Linking Format in UNIX for C99

Note that the UNIX prompt depends on the shell being running. It can be `$` or `%` or any other customized prompt. The command `c99` invokes the C compiler with the C99 standard implementation (other versions are usually available for backward compatibility). The option can be either `-c` for compiling only, or `-o` for either linking or compiling and linking. The `-o` option indicates that we need an executable module. The intent of the `-o` option is conveyed through the parameters:

1. If we want to compile only, we define only source files; the executable file name is not used.
2. If we want to link only, we define only the object files; the source files are not used.

Examples

Let's look at some examples.

EXAMPLE K-1 *Compile, Link, and Execute:* We need to compile and link to create an executable file from a source file named `myprogram.c`. The following code shows the commands. In the first line, we create an executable file named `myprogram`; in the second line we run that program. After these two lines of commands have been executed, we have two files in our directory: `myprogram.c` and `myprogram`.

```
$ c99 -o myprogram myprogram.c
$ myprogram
```

EXAMPLE K-2 *Compile Two Source Files:* We need only to compile two files called `first.c` and `second.c`. When the compiler completes, and assuming no errors, we have four files in our directory: `first.c`, `first.o` (object file), `second.c`, and `second.o`.

```
$ c99 -c first.c second.c
```

Of course, we could use two commands instead of one and compile each file separately.

EXAMPLE K-3 *Create Executable File:* We need only to create an executable file from the `first.o` file (linking all library files to it). After the linker executes, we have a executable new file in our directory, `first`.

```
$ c99 -o first first.o
```

EXAMPLE K-4 *Create Executable from Two Object Files:* We need to link two object files, `first.o` and `second.o`, together to create an executable file `firstSecond`. The following shows the command.

```
$ c99 -o firstSecond first.o second.o
```

EXAMPLE K-5 *Separate Compilation:* In separate compilations, we separately compile the source files `myprogram.c` and `power2.c` and link them to create an executable file `mybigprogram`. We then run the executable file.

```
$ c99 -c myprogram.c
$ c99 -c power2.c
$ c99 -o mybigprogram myprogram.o power2.o
$ mybigprogram
```

EXAMPLE K-6 *Compile and Execute Short Cut:* Sometimes, programmers use a short cut that does not save the executable file. If we do not define the name of the executable file, the system creates a file called `a.out`. For example, the following command shows how we can compile, link, and run a source file named `temporary.c`.

```
$ c99 temporary.c
$ a.out
```

Note that because the name of the file is system defined, it remains in the system until we created another executable file, which the system needs to call `a.out`. This method is used only for temporary testing and is not recommended when we need to keep the executable file.

Other Platforms

In general, every hardware platform comes with its own application development environment. In addition, there are numerous compilers and development support tools available. To find the documentation for your local support environment, ask your support staff or check with the help system installed with your compiler.

Appendix L
Understanding Complex Declarations

The declarations have become increasingly more complicated throughout this text. Sometimes the declarations are difficult to interpret, even for someone experienced in the C language.

To help you read and understand complicated declarations, we provide a rule that we call the **right-left rule**. Using this rule to interpret a declaration, you start with the identifier in the center of a declaration and read the declaration by alternatively going right and then left until you have read all entities. Figure L-1 is a representation of the basic concept.

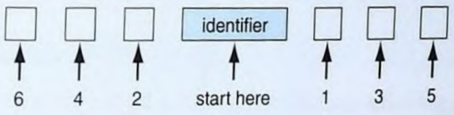


FIGURE L-1 Right–Left Rule Concept

We will begin with some simple examples and proceed to the more complicated.

1. Consider the simple declaration

```
int x
```

This is read as "x is # an integer."[1]

```
int    x     #
 ↑     ↑     ↑
 2     0     1
```

Since there is nothing on the right, we simply go left.

2. Now consider the example of a pointer declaration. This example is read as "p is # a pointer # to integer."

```
int    *     p     #     #
 ↑     ↑     ↑     ↑     ↑
 4     2     0     1     3
```

Note that we keep going right even when there is nothing there until all the entities on the left have been exhausted.

3. In the next example, we have an equal number of entities on the right and the left.

```
int    table  [4]
 ↑       ↑     ↑
 2       0     1
```

This declaration is read as "table is an array of 4 integers."

4. Regardless of how many dimensions are in an array, it is considered as one element in the rule. Therefore, given the following declaration of a multidimensional array:

```
int    table  [4][5]
 ↑       ↑      ↑
 2       0      1
```

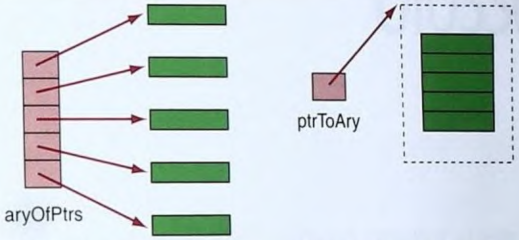
it is read as "table is a [4][5] array of integers."

5. The next example is quite difficult and is often misread. In this declaration, we have an array of pointers to integers. The structure is seen in Figure L-2a.

```
int  *  aryOfPtrs [5] #
 ↑   ↑      ↑      ↑  ↑
 4   2      0      1  3
```

It is read as "`aryOfPtrs` is an array of 5 pointers to # integer."

[1] The pound sign (#) is just a placeholder to show that there is no entity to be considered. It is ignored when read.



(a) An array of pointers

(b) A pointer to an array

FIGURE L-2 Array of Pointers Versus Pointer to Array

6. By using parentheses, we change the previous example to a pointer to an array of five integers. In this case, the pointer is to the whole array, not just one element in it. (See Figure L-2b.)

```
int (*ptrToAry #) [5]
 ↑   ↑   ↑     ↑   ↑
 4   2   0     1   3
```

This declaration is read "`ptrToAry` is # a pointer to an array of 5 integers."

7. This example deals with function declarations. Here, we see a simple prototype for a function that returns an integer.

```
int doIt (…)
 ↑    ↑   ↑
 2    0   1
```

This declaration is read as "`doIt` is a function returning an integer."

8. The final example shows a function returning a pointer to an integer.

```
int * doIt (int) #
 ↑  ↑   ↑    ↑   ↑
 4  2   0    1   3
```

This example is read "`doIt` is a function returning a pointer to # an integer."

Glossary

A

absolute value: the magnitude of a number regardless of its sign. In C: *abs*.

abstract data type (ADT): a data declaration and a set of operators that are allowed on the data, encapsulated as a type.

accuracy: the quality factor that addresses the correctness of a system.

actual parameter: the parameter in the function calling statement that contains the values to be passed to the function. Contrast with *formal parameter*.

additive expression: in C, the binary addition and subtraction expressions.

address list: in C: function parameter list that consists of variable addresses.

address operator: in C: the ampersand (`&`).

afferent module: a module whose processing is directed toward the central transform; that is, a module that gathers data to be transmitted toward the central processing functions of a module.

algorithm: the logical steps necessary to solve a problem in a computer; a function or a part of a function.

algorithmics: the term created by Brassard and Bratley that refers to the study of techniques used to create efficient algorithms.

alternate flag: the data formatting flag used with real, engineering, hexadecimal, and octal conversion codes indicating that an alternative presentation should be used.

ANSI/ISO C: the standard for the C language adopted by the American National Standards Institute.

append mode: in file processing, the mode that adds to the end of a file.

application software: computer software developed to support a specific user requirement. Contrast with *system software*.

application-specific software: any application software that can be used for only one purpose, suc as an accounting system.

arc: a directed line in a graph; contrast with *edge*.

array: a sequenced collection of elements of the same data type.

array of pointers: an array of addresses, often us to represent a sparse array.

ascending sequence: a list order in which eac element in the list has a key greater than or equal its predecessors. Contrast with *descending sequer*

ASCII: the American Standard Code for Informat Interchange. An encoding scheme that defines con characters and graphic characters for the first 128 ues in a byte. The first 128 characters in Unicode.

assembler: system software that converts a sour program into executable object code; traditionally associated with assembly language program. See *compiler*.

assembly language: a programming language i which there is a one-for-one correspondence between the computer's machine language and t symbolic instruction set of the language.

assignment expression: an expression contain the assignment operator (=) that results in the va of the expression being placed into the left opera

associativity: the parsing direction used to eval an expression when all operators have an equal ity. See *left-to-right* and *right-to-left associativity*. also *precedence*.

atomic data: data that cannot be meaningf subdivided.

atomic data type: a set of atomic data having identical properties that distinguish one type from another. Contrast with *derived type*.

***auto*:** the default storage class for a local variable.

auxiliary storage: any storage device outside main memory; permanent data storage; external storage.

B

base case: in recursion, the event that does not involve a recursive call and therefore ends the recursion.

batch update: an update process in which transactions are gathered over time for processing as a unit. Contrast with *online update*.

bi-directional communication: communication between a calling and called function in which data flows in both direction, that is, from the calling function to the called function and then from the called function to the calling function.

big-O analysis: the analysis of an algorithm that measures efficiency using big-O notation rather than mathematical precision.

big-O notation: a measure of the efficiency of an algorithm in which only the dominant factor is considered.

binary: a numbering system with base 2. Its digits are 0 1.

binary expression: any expression containing one operator and two operands.

binary file: a collection of data stored in the internal format of the computer. Contrast with *text file*.

binary search: a search algorithm in which the search value is located by repeatedly dividing the list in half.

binary search tree: A binary tree in which: (1) the keys in the left subtree are all less than the root key, (2) the keys in the right subtree are greater than or equal to the root key, and (3) the subtrees are all binary search trees.

binary stream: a stream in which data is represented as it is stored in the memory of a computer.

binary tree: a tree in which no node can have more than two children.

bit: acronym for Binary digIT. The basic storage unit in a computer with the capability of storing only the values 0 or 1.

bitwise operator: any of the set of operators that operate on individual bits.

blackbox testing: testing based on the system requirements rather than a knowledge of the program. Contrast with *whitebox testing*.

block: in C, a group of statements enclosed in braces `{…}`.

block comment: a comment that can span one or more lines. Contrast with *line comment*.

block scope: see *scope*.

body of loop: the code executed during each iteration of a loop.

Boolean: a variable or expression that can assume only the values true or false.

braces: the `{` and `}` symbols.

breadth-first traversal: a graph traversal in which nodes adjacent to the current node (siblings) are processed before their descendents. Contrast with *depth-first traversal*.

break: a C statement that causes a *switch* or loop statement to terminate.

bubble sort: a sort algorithm in which each pass through the data moves (bubbles) the lowest element to the beginning of the unsorted portion of the list.

buffer: (1) hardware, usually memory, used to synchronize the transfer of data to and from main memory. (2) memory used to hold data that have been read before they are processed, or data that are waiting to be written.

buffered input/output: input or output that are temporarily stored in intermediate memory while being read or written.

buffered stream: C term for buffered input/output.

bug: a colloquial term used for any error in a piece of software.

byte: a unit of storage, 8 bits.

C

C standard library: any of the libraries that contain predefined algorithms delivered with the system, such as standard library, (`stdlib.h`) and standard input and output (`stdio.h`).

called function: in a function call, the function to which control is passed.

calling function: in a function call, the function that invokes the call.

cast: a C operator that changes the type of an expression.

ceiling: the smallest integral value that is larger than a floating-point value.

central processing unit (CPU): the part of a computer that contains the control components—that is, the part that interprets instructions. In a personal computer, a microchip containing a control unit and an arithmetic-logical unit.

central transform: the modules of a program that take input and convert it to output. See also *afferent module* and *efferent module*.

changeability: the quality factor that addresses the ease with which changes can be accurately made to a program.

character: a member of the set of values that are used to represent data or control operations. See *Unicode*.

character constant: a constant that defines one character enclosed in two single quotes.

character set: the set of values in a computer's alphabet. See *Unicode*.

child: a node in a tree or a graph that has a predecessor.

chronological list: a list that is organized by time; that is, in which the data are stored in the order in which they were received. See also *FIFO* and *LIFO*.

classifying function: a standard C character function that classifies a character according to C's character taxonomy, such as "printable" or "alphanumeric."

clear box testing: see *whitebox testing*.

client: in a client/server network, the computer that provides the basic application computing; the computer residing in the user's physical area.

client/server: a computer system design in which two separate computers control the processing of the application, one providing the basic application computing (the client) and the other providing auxiliary services, such as database access (the server).

close function: the function that concludes the writing of a file by writing any pending data to the file and then making it unavailable for processing.

cohesion: the attribute of a module that describes how closely the processes within a module are related to each other.

coincidental cohesion: the cohesion level in which totally unrelated processes are combined into a module.

column: in a two-dimensional table, the dimension representing vertical columns in the array. Contrast with *row*.

comma expression: the last (lowest precedence) expression in C; used to combine multiple expressions into one expression statement.

comma operator: in C, the operator that connects multiple statements in one expression.

command-line argument: in C, a parameter specified on the run-time execute statement and passed to the program for its use.

comment: in a C program, a note to the program reader that is ignored by the compiler.

communicational cohesion: the processes in a module are related only in that they share the same data.

compilation: the process used to convert a source program into machine language.

compiler: system software that converts a source program into executable object code; traditionally associated with high-level languages. See also *assembler*.

complement: in a set of values with two symmetric sections, the operation that gives the counterpart of the current value, that is, the value in the other section. For example, the complement of a number is a number with the same value and the sign reversed. The complement of a binary digit is the other binary digit. The complement of a logical value (*true* or *false*) is the other value.

complex type: a data type consisting of a real and an imaginary type and representing a complex number in mathematics.

composite data: data that are built on other data structures; that is, data that can be broken down into discrete atomic elements.

compound statement: a sequence of statements enclosed in braces. See also *block*.

computer hardware: the set of computer components, consisting of at least an input device, an output device, a monitor, and a central processing unit.

computer language: any of the syntactical languages used to write programs for computers, such as machine language, assembly language, C, COBOL, and FORTRAN.

computer software: the set of programs, consisting of system and application programs, that provide instructions for the computer.

computer system: the set of computer hardware and software that make it possible to use a computer.

conditional expression: in C, shorthand form of a two-way selection.

connected graph: a graph is connected if, when direction is suppressed, there is a path from any vertex to any other vertex.

***const*:** a storage class that designates that a field's contents cannot be changed during the execution of the program.

constant: a data value that cannot change during the execution of the program. Contrast with *variable*.

content coupling: the direct reference to the data in one module by statements in another module. The lowest form of coupling and one to be avoided.

continue: in C, a statement that causes the remaining code in a loop iteration to be skipped.

control character: a nonprintable character value whose function is to perform some operation, such as form-feed, or that is used to indicate status, such as the start of a transmission.

control coupling: communication between functions in which flags are set by one module to control the actions of another.

conversion code: in formatted input and output, the code in the format specification that identifies the data type.

conversion specification: the specification that defines how data are to be represented in an input or output operation.

converting function: a C character function that converts lowercase alphabetic characters to uppercase, or vice versa.

correctability: the quality factor that addresses the ease with which errors in a module can be fixed.

counter-controlled loop: a looping technique in which the number of iterations is controlled by a count; in C, the *for* statement. Contrast with *event-controlled loop*.

coupling: a measure of the interdependence between two separate functions. See also: *content coupling*, *control coupling*, *data coupling*, *global coupling*, and *stamp coupling*.

CPU: see central processing unit.

D

dangling else: a code sequence in which there is no *else* statement for one of the *if* statements in a nested *if*.

data: the part of a structure that stores application information.

data coupling: communication between modules in which only the required data are passed. Considered the best form of coupling.

data encapsulation: the software engineering concept in which data are hidden from an application program's access.

data flow: in a structure chart, an indicator identifying data input to or output from a module.

data hiding: the principle of structured programming in which data are available to a function only if it needs them to complete its processing; data not needed are "hidden" from view. See also *encapsulation* and *scope*.

data name: an identifier given to data in a program.

data structure: the syntactical representation of data organized to show the relationship among the individual elements.

data type: a named set of values and operations defined to manipulate them, such as character and integer.

data validation: the process of verifying and validating data read from an external source.

declaration: in C, the association of a name with an object, such as a type, variable, structure, or function. See also *definition*.

declaration section: the portion of a block in which variables and constants may be defined. The first section in a block. See also *statement section*.

default: in C, the entry point to the code that is to be executed if none of the case values match the *switch* expression.

define: in C, a preprocessor command that names and provides the code for a macro.

definition: in C, the process that reserves memory for a named object, such as a variable or constant. See also *declaration*.

degree: the number of lines incident to a node in a graph.

delimited string: a string terminated by a nondata character, such as the null character in C.

delimiter: the character or token that identifies the end of a structure.

De Morgan's rule: a rule used to complement a logical expression.

demotion: an implicit type conversion in which the rank of an expression is temporarily reduced to match other elements of the expression.

dependent statement: a program statement, such as the true action in an *if* statement, whose execution is controlled by another statement.

depth-first traversal: a traversal in which all of a node's descendents are processed before any adjacent nodes (siblings). Contrast with *breadth-first traversal*.

dequeue: delete an element from a queue.

dereference: access of a data variable through a pointer containing its address.

dereference operator: in C, the asterisk (*). Used to indicate that a pointer's contents (an address) are to be used to access another piece of data.

derived type: a composite data type constructed from other types (array, structure, union, pointer, and enumerated type). Contrast with *atomic type*.

descendant: any node in a path from the current node to a leaf.

descending sequence: a list order in which each element in a list has a key less than or equal to its predecessor.

directed graph: a graph in which direction is indicated on the lines (arcs).

disk: an auxiliary storage medium used to store data and programs required for a computer.

disk drive: the auxiliary storage hardware device used to read and write a disk.

diskette: a removable flexible disk, enclosed in a protective flexible or rigid cover, used to store data for a personal computer.

distributed environment: an environment that provides a seamless integration of computing functions between different servers and clients.

***do...while* loop:** a sentinel-controlled, post-test loop in C.

double: the C type for double-precision floating point type.

downward communication: data flow from the calling function toward the called function.

drive: an auxiliary storage device that can write and read data, such as the internal hard disk, a floppy disk, or a tape unit.

dynamic allocation: allocation of memory for storing data during the execution of a program. Contrast with *static memory allocation*.

dynamic array: an array that has been allocated in the heap during the execution of the program.

dynamic memory: memory whose use can change during the execution of the program (The heap).

E

edit set: a list of characters used to validate characters during the formatted reading of a string.

efferent module: a module whose processing is directed away from the central transform; that is, a module that predominately disposes of data by reporting or writing to a file.

efficiency: the quality factor that addresses the optimum use of computer hardware or responsiveness to a user.

edge: a line in an undirected graph. Contrast with *arc*.

***else-if*:** a style (as opposed to syntax) convention used to implement a multiway selection for a nonintegral expression. Each expression in the series must evaluate the same variable.

empty list: a list that has been allocated but that contains no data. Also known as a null list.

encapsulation: the software engineering design concept in which data and their operations are bundled together and maintained separately from the application using them.

end of file: the condition that occurs when a read operation attempts to read after it has processed the last piece of data.

enumeration constant: the identifier associated with a value in an enumeration declaration.

enqueue: insert an element into a queue.

EOF: end of file. In C, a flag set to indicate that a file is at the end.

error report file: in a file update process, a report of errors detected during the update.

error state: one of three states that an open file may assume. An error state occurs when a program issues an impossible command, such as a read command while the file is in a write state, or when a physical device failure occurs. See also *read state, write state*.

error stream: in C, the standard file used to display errors, often assigned to a printer (*stderr*).

escape character: in C, the backslash (\) character used to identify a special interpretation for the character that follows it.

escape sequence: any group of keystrokes or characters used to identify a control character. In C, the backslash (\) used to indicate that the character that follows represents a control character.

event-controlled loop: a loop whose termination is predicated upon the occurrence of a specified event. Contrast with *counter-controlled loop*.

exchanging values: the logic used to exchange the values in two variables.

***exclusive or*:** a binary logical operation in which the result is true only when one of the operands is true and the other is false.

executable file: a file that contains program code in its executable form; the result of linking the source code object module with any required library modules.

explicit type conversion: the conversion of a value from one type to another through the cast operator. Contrast with *implicit type conversion*.

exponential efficiency: a category of program/module efficiency in which the run time is a function of the power of the number of elements being processed.

expression: a sequence of operators and operands that reduces to a single value.

expression statement: an expression terminated by a semicolon.

expression type: one of six expression attributes that broadly describes an expression's format: *primary, postfix, prefix, unary, binary,* and *ternary*.

***extern*:** the storage class that specifies that an object has static extent and that its name is exportable to the linker.

F

factorial efficiency: a measure of the efficiency of a module in which the run time is proportionate to the number of elements factorial.

false: the Boolean value used to indicate that an expression is not true.

fan out: an attribute of a module that describes the number of submodules it calls.

field: the smallest named unit of data that has meaning in describing information. A field may be either a variable or a constant.

field specification: a subcomponent of the format string used to describe the formatting of data, in the formatted input and output functions.

field width: in a field specification, the specification of the maximum input width or minimum output width for formatted data.

FIFO: first in-first out.

file: a named collection of data stored on an auxiliary storage device. Compare with *list*.

file mode: a designation of a file's input and/or output capability; files may be opened for reading, writing, appending, or updating.

file state: the operating condition of a file: read state, write, or error state.

file table: in C, the predefined standard structure, `FILE`, used to store the attributes of a file.

filename: the operating system name of a file on an auxiliary storage device.

first in-first out: a data structure processing sequence in which data are processed in the order that they are received; a *queue*.

fixed-length array: an array in which the number of elements are predetermined during the declaration/definition.

fixed-length string: a string whose size is constant regardless of the number of characters stored in it.

flag: an indicator used in a program to designate the presence or absence of a condition; *switch*.

flag modifier: the input/output format specification that suppresses input or modifies output formatting.

flexibility: the quality factor that addresses the ease with which a program can be changed to meet user requirements.

flipping bits: changing a bit from a 0 to 1 or a 1 to 0.

floating-point number: a number that contains both an integral and a fraction.

floor: the largest integral number smaller than a floating-point value.

flowchart: a program design tool in which standard graphical symbols are used to represent the logical flow of data through a function.

***for* loop:** a counter-controlled, pretest loop in C.

force to change: the bit manipuation concept used to force selected bits to change from 0 to 1 or from 1 to 0.

force to one: the bit manipulation concept used to force selected bits to 1.

force to zero: the bit manipulation concept used to force selected bits to zero.

formal parameter: the parameter declaration in a function prototype used to describe the type of data to be processed by the function. Contrast with *actual parameter.*

formal parameter list: the declaration of parameters in a function header.

format control string: in a formatted input/output function, the string that is used for formatting data.

formatted input/output: in C, any of the standard library functions that can reformat data to and from text while they are being read or written.

frequency array: an array that contains the number of occurrences of a value or of a range of values. See also *histogram.*

front: when used to refer to a list, a pointer or index that identifies the first element.

function: a named block of code that performs a process within a program; an executable unit of code, consisting of a header, function name, and a body, that is designed to perform a task within the program.

function body: the part of a function that contains the local definitions and statements; all of a function except the header declaration. Contrast with *function header.*

function call: an expression that involves the execution of a function.

function declaration: in C, statement that describes a function's return type, name, and formal parameters.

function definition: in C, the implementation of a function declaration.

function header: in a function definition, that part of the function that supplies the return type, function identifier, and formal parameters. Contrast with *function body.*

functional cohesion: a module in which all of the processing is related to a single task. The highest level of cohesion.

G

general case: all the code in a recursive function that is not a base case.

general-purpose software: software, such as a spreadsheet, that can be used in multiple applications.

global coupling: communication between different modules that uses data accessible to all modules in a program. Considered to be a very poor communications technique for intraprogram communication.

global declaration section: the declaration and/or definition of a variable or function outside the boundaries of any function—that is, before *main* or between function definitions. Contrast with *local declaration.*

global variable: a variable defined in the global declaration section of a program—that is, defined outside a function block.

graph: a non-linear list in which each element can have zero, one, or more predecessors and zero, one, or more successors.

H

hard copy: any computer output that is written to paper or other readable mediums such as microfiche. Contrast with *soft copy.*

hardware: any of the physical components of a computer system, such as the keyboard or a printer.

header declaration: that part of a function that contains the return type, function name, and parameter declarations.

ader file: in C, a file consisting of prototype ements and other declarations and placed in a ary for shared use.

ad pointer: a pointer that identifies the first elent of a list.

ap: see *heap memory.*

ap memory: a pool of memory that can be used lynamically allocate space for data while the prom is running.

xadecimal: a numbering system with base 16. Its its are 0 1 2 3 4 5 6 7 8 9 A B C D E F.

gh-level language: a (portable) programming guage designed to allow the programmer to conntrate on the application rather than the structure a particular computer or operating system.

stogram: a graphical representation of a freency distribution. See also *frequency array.*

I

entifier: the name of an object. In C, identifiers n consist only of digits, letters, and the underscore.

.. *else*: a decision statement in C.

aginary type: the data type in C that can repreent and imaginary number in mathematics.

plicit type conversion: the automatic conversion data from one type to another when required within C program. Contrast with *explicit type conversion.*

clude: in C, a preprocessor command that speciss a library file to be inserted into the program.

degree: in a tree or graph, the number of lines itering a node.

dentation: a coding style in which statements ependent on a previous statement, such as *if* or *hile,* are coded in an indented block to show their lationship to the controlling statement.

dex: the address of an element within an array. ee also *subscript.*

dex range checking: a feature available in some ompilers that inserts code to ensure that all index eferences are within the array.

dexing: a process used to identify one element in n array.

ndirect pointer: a pointer that locates the address f data through one or more other pointers; pointer to ointer.

infinite loop: a loop that does not terminate.

information hiding: a structured programming concept in which the data structure and the implementation of its operations are not known by the user.

initialization: the process of assigning values to a variable at the beginning of a program or a function.

initializer: in C, any value assigned to a constant or variable when it is defined.

inorder traversal: a binary tree traversal in which the root is processed after the left subtree and before the right subtree. Contrast with *preorder* and *postorder traversal.*

input device: a device that provides data to be read by a program.

input stream: in C, the flow of data from a file to the program.

inquiry: a request for information from a program.

inquiry all: an algorithm that determines if all data items in a list meet some criteria.

inquiry any: an algorithm that determines if any data item in a list meet some criteria.

insertion sort: a sort algorithm in which the first element from the unsorted portion of the list is inserted into its proper position relative to the data in the sorted portion of the list.

integer: an integral number; a number without a fractional part.

integral type: a data type that can store only whole numbers.

intelligent data name: a software engineering principle that requires the identifier to convey the meaning or use of an object.

interoperability: the quality factor that addresses the ability of one system to exchange data with another.

iteration: a single execution of the statements in a loop.

J

justification: an output formatting parameter that controls the placement of a value when it is shorter than the specified output width.

K

key: one or more fields that are used to identify a record (structure).

keyboard: an input device used for text or control data, that consists of alphanumeric keys and function keys.

key-sequence list: a list in which the data items are ordered based on the value of a key.

keywords: see *reserved words*.

KISS: in this text: Keep It Short and Simple.

L

last in-first out: a data structure processing sequence in which data are processed in the reverse order that are received; a *stack*.

Latin character set: the extended ASCII character set in Unicode.

leading zero flag: the flag in the format string of a print statement indicating that numeric data are to be printed with leading zeros.

leading zero padding: adding extra zeros at the beginning of a data item.

left justification: the orientation of variable-length data in an output format such that trailing null values are inserted and the first data character is at the left end of the print area. Contrast with *right justification*.

left-to-right associativity: the evaluation of an expression that parses from the left to the right. Contrast with *right-to-left associativity*.

length-controlled string: a variable-length string function in which the data are identified by a structural component containing the length of the data.

lexicographical: a textual data order based on the dictionary.

LIFO: last in-first out.

limit test expression: in a loop, the expression that determines if the loop will continue or stop. See also *terminating condition*.

linear efficiency: a measure of the efficiency of a module in which the run time is proportionate to the number of elements being processed.

linear list: a list structure in which each element, except the last, has a unique successor.

linear loop: a loop whose execution is a function of the number of elements being processed. See also *linear efficiency*.

line comment: a comment that spans only to the end of the current line. Contrast with *block comment*.

linear search: see *sequential search*.

link: in a list structure, the field that identifies the next element in the list.

linked list: a linear list structure in which the ordering of the elements is determined by link fields.

linked list traversal: processing in which every element of a linked list is processed in order.

linker: the program creation process in which an object module is joined with precompiled functions to form an executable program.

list: an ordered set of data contained in main memory. Compare with *file*.

list traversal: any logic that visits and processes all the data in a list.

literal: an unnamed constant coded in an expression.

loader: the operating system function that fetches an executable program into memory for running.

local declaration: a variable or type declaration that is only visible to the block in which it is contained. Contrast with *global declaration*.

local variable: a variable defined with a block.

logarithmic efficiency: a measure of the efficiency of a module in which the run time is proportionate to the log of the number of elements being processed.

logarithmic loop: a loop whose efficiency is a function of the log of the number of elements being processed. See also *logarithmic efficiency*.

logical cohesion: a design attribute that describes a module in which the processing within the module is related only by the general type of processing being done. Considered unacceptable design in structured programming.

logical data: data whose values can be only true or false. See *Boolean*.

logical operator: a C operator (`&&`, `||`, `!`) used in a logical expression.

loop: the construct used to repeat one or more statements in a program. The *while*, *for*, and *do...while* statements in C.

p control expression: the expression in a loop ement that is used to determine if the body of the is to be executed.

p update: the code within a loop statement or ly that changes the environment such that the p will eventually terminate.

lue: an expression that allows the contents of a iable to be modified.

M

chine language: the instructions native to the ntral processor of a computer and that are execut- le without assembly or compilation.

ain memory: see *memory.*

ask: a variable or constant that contains a bit con- uration used to control the setting of bits in a bit- se operation.

aster file: a permanent file that contains the most rrent data regarding an application.

emory: the main memory of a computer consist- g of random access memory (RAM) and read-only emory (ROM); used to store data and program structions.

emory constant: a C type defined using the type ualifier *const.*

erge: to combine two or more sequential files into ne sequential file based on a common key and ructure format.

etadata: data about a list or other data structure ored within the data structure itself.

onitor: the visual display unit of a computer sys- em, usually a video display device.

ultidimensional array: an array whose elements onsist of one or more arrays.

ultiplicative expression: an expression that con- ins a multiply, divide, or modulus operator.

ultiway selection: a selection statement that is apable of evaluating more than two alternatives. In , the *switch* statement. Contrast with *two-way election*.

N

atural language: any spoken language.

egative logic: an expression that begins with the egation operator (`!`).

nested *if* statement: an *if* statement coded as either the true or false statement within another *if*.

nested loop: a loop contained within another loop.

nested structure: a structure that contains other structures.

node: in a data structure, an element that contains both data and structural elements used to process the list.

nonlinear list: a list in which each element can have more than one successor.

null *else*: the absence of a false statement in an *if* statement.

O

object module: the output of a compilation consisting of machine language instructions.

octal: a numbering system with a base of 8. The octal digits are 0 1 2 3 4 5 6 7.

offset: in pointer arithmetic, a computed value that is used to determine another index location within an array.

one-dimensional array: an array with only one level of indexing.

one's complement: the bitwise operator that reverses the value of the bits in a variable.

online update: an update process in which transactions are entered and processed by a user who has direct access to the system. Contrast with *batch update*.

open function: the function that locates and prepares a file for processing.

operability: the quality factor that addresses the ease with which a system can be used.

operand: an object in a statement on which an operation is performed. Contrast with *operator.*

operating system: the software that controls the computing environment and provides an interface to the user.

operator: the syntactical token representing an action on data (the operand). Contrast with *operand.*

ordered list: a list in which the elements are arranged so that the key values are placed in ascending or descending sequence.

outdegree: the number of lines leaving a node in a tree or a graph.

output device: a device that can be written but not read.

output stream: in C, the flow of data from a program to the file.

overflow: the condition that results when an attempt is made to insert data into a structure and there is no room.

P

padding: extra zeros or spaces added to the left or right of a data item.

parameter: a value passed to a function.

parameter list: a list of values passed to a function. (Note: The values may be data or addresses.)

parent: a tree or graph node with an outdegree greater than 0; that is, with successor.

pass by address: in C, the technique used when a pointer is passed to a function as a parameter.

pass by reference: a parameter passing technique in which the called function refers to a passed parameter using an alias name.

personal computer (PC): a computer designed for individual use.

plane: the third dimension in a multidimensional array.

pointer: a constant or variable that contains an address that can be used to access data.

pointer arithmetic: addition or subtraction in which a pointer's contents (an address) are changed by a multiple of the size of the data to which it is pointing.

pointer constant: a pointer whose contents cannot be changed.

pointer indirection: the use of a pointer to access data.

pointer to function: a pointer that identifies the entry point to a function. It is used to pass a function's address as a parameter.

pop: the stack delete operation.

polyonymic efficiency: a measure of the efficiency of a module in which the run time is proportionate to the number of elements raised to the highest factor in a polynomial.

portability: the quality factor that addresses the ease with which a system can be moved to other hardware environments.

postfix decrement: in C, the operator (such as a– –) that subtracts 1 from a variable after its value has been used in an expression.

postfix expression: an expression in which the operator follows the operand.

postfix increment: in C, the operator (such as a++) that adds 1 to a variable after its value has been used in an expression.

post-test loop: a loop in which the terminating condition is tested only after the execution of the loop statements. Contrast with *pretest loop*.

precedence: the priority assigned to an operator or group of operators that determines the order in which operators will be evaluated in an expression. See also *associativity*.

precision: in the format string of a print function, the maximum number of integral digits, the number of significant digits or fractional digits in a floating-point number, or the maximum number of characters in a string.

precision modifier: the print format string modifier that specifies the number of decimal places to be printed in a floating-point value.

prefix decrement: in C, the operator (such as --a) that subtracts one from a variable before its value is used in an expression. Also known as *unary decrement*.

prefix expression: an expression in which the operator precedes the operand.

prefix increment: in C, the operator (such as ++a) that adds 1 to a variable before its value is used in an expression. Also known as *unary increment*.

preprocessor: the first phase of a C compilation in which the source statements are prepared for compilation and any necessary libraries are loaded.

preprocessor directives: commands to the C precompiler.

pretest loop: a loop in which the terminating condition is tested before the execution of the loop statements. Contrast with *post-test loop*.

primary expression: an expression consisting of only a single operator; the highest priority expression.

primary storage: the volatile (RAM) memory of a computer.

intable character: a character value that is associated with a print graphic.

inter: an output device that displays the output n paper.

rocedural cohesion: a module design in which e processing within the module is related by control flows. Considered acceptable design only at the igher levels of a program.

rocess-control loops: a continuous loop used to ontrol operating equipment, such as a building air onditioning/heating system.

rocessing unit: see *central processing unit*.

rogram development: the system development ctivity in which requirement specifications are conerted into executable programs.

rogram file: a file that contains an executable rogram.

rogram testing: the process that validates a program's operation and verifies that it meets its design equirements.

rototype declaration: the function header that leclares the return type, function name, and parameter number and types without defining the function tself.

program documentation: comments placed within a program to help the reader understand the purpose of the program or a portion of its implementation.

promotion: an implicit type conversion in which the rank of an expression is temporarily increased to match other elements of the expression.

prototype statement: in C, the declaration of a function that provides the return type and formal parameter types.

pseudocode: English-like statements that follow a loosely defined syntax and are used to convey the design of an algorithm or function.

push: the stack insert operation.

Q

quadratic efficiency: a measure of the efficiency of a module in which the run time is proportionate to the number of elements squared. Quadratic efficiency is one of the polyonymic factors.

quadratic loop: a loop that consists of two or more loops, each of which have a linear efficiency, resulting in a loop with quadratic efficiency.

quality: see *software quality*.

query: see *inquiry*.

queue: a linear list in which data can be inserted only at one end, called the *rear*, and deleted from the other end, called the *front*.

R

ragged array: a two-dimensional array in which the number of elements in the rows is not equal.

random number: a number selected from a set in which all members have the same probability of being selected.

random number seed: the initial number passed to a random-number generator.

read mode: the attribute of a file that indicates that it is opened for input only.

read state: one of three states that an open file may assume. The state of a file during which only input operations may be performed. See also *error state, write state*.

real time: processing in which updating takes place at the time the event occurs.

real type: a data type representing a number with a fraction.

rear: when used to refer to a list; a pointer that identifies the last element.

record: see *structure*.

recursion: a function design in which the function calls itself.

register: the storage class that requests that, if possible, a variable be allocated to a CPU register.

reliability: the quality factor that addresses the confidence or trust in a system's total operation.

reserved words: the set of words in C that have a predetermined interpretation and cannot be used in the definition of an object.

return: the C statement that causes execution of a function to terminate and control to be resumed by the calling function or the operating system.

reusability: the quality factor that addresses the ease with which software can be used in other programs.

reusable code: code that can be used by more than one process.

rewind: the C function used to position the file marker at the beginning of an open file.

right-to-left associativity: the evaluation of an expression that parses from the right to the left. Contrast with *left-to-right associativity.*

right justification: the orientation of variable length data in an output format such that leading null values are inserted and the last data character is at the right end of the print area. Contrast with *left justification.*

right–left rule: a method of reading complex declarations or definitions that starts with the identifier and alternately reads right and left until the declaration has been fully read.

row: in a two-dimensional array, elements along a horizontal axis. Contrast with *column.*

root: the first node of a tree; the node with no predecessor.

rvalue: an expression attribute indicating that the expression can be used only to supply a value for another expression.

S

scanset: the set of allowed or disallowed characters when reading a string.

scope: an attribute of a variable that defines whether it is visible to or hidden from statements in a program. See also *temporal authority.*

search: the process that examines a list to locate one or more elements containing a designated value known as a search argument.

secondary storage: see *auxiliary storage.*

security: the quality factor that addresses the ease or difficulty with which an unauthorized user can access data.

seed: the variable used in a random number generator to calculate the next number in the series. In C, the seed is set by the library function *srand.*

seek file: in the processing of a disk file, the action that moves the index marker to a new position in the file; in C, the *fseek* function.

selection sort: the sort algorithm in which the smallest value in the unsorted portion of a list is selected and placed at the end of the sorted portion of the list.

selection statement: a statement that chooses between two or more alternatives. In C, the *if...else* or *switch* statements.

self-referential structure: a structure that contains a pointer to itself.

sentinel: a flag that guards the end of a list or a file. The sentinel is usually the maximum value for a key field and cannot be a valid data value.

separate compilation: a capability within a programming language that allows parts of a program to be compiled independently and later assembled into an executable unit.

sequential cohesion: a module design in which the processing within the module flows such that the data from one process are used in the next process.

sequential file: a file structure in which data must be processed serially from the first entry in the file.

sequential search: a search technique used with a linear list in which the searching begins at the first element and continues until the value of an element equal to the value being sought is located, or until the end of the list is reached.

serial search: any search technique that starts at the beginning of the list and continues toward the end.

server: in a client/server system, the centralized computer that provides auxiliary services.

***short*:** the C type for short integer.

***short integer*:** an integer format used for small numbers, generally less than 32,768.

side effect: a change in a variable that results from the evaluation of an expression; any input/output performed by a called function.

sign flag: the flag value (+) in the conversion code of a print format string that indicates that positive numbers must be printed with a plus sign.

signed: a type modifier indicating that a numeric value may be either positive or negative. Contrast with *unsigned.*

size: an indicator (`h`, `l`, `L`) in the conversion code of a format string that modifies integer and *float* types.

size modifier: the data formatting token that modifies the conversion code.

slack bytes: inaccessible memory locations added between fields in a structure to force a hardware-required boundary alignment.

soft copy: computer output written to a nonpermanent display such as a monitor. Contrast with *hard copy.*

software: the application and system programs necessary for computer hardware to accomplish a task including their documentation and any required procedures.

software quality: an attribute of software that measures the user's total satisfaction with a system.

sort: the process that orders a list or file.

sort pass: one sort loop during which all elements are tested.

source file: the file that contains program statements written by a programmer before they are converted into machine language; the input file to an assembler or compiler.

space flag: the flag value in the conversion code of a print format string indicating that positive numbers must be printed with a leading space.

sparse array: a two-dimensional array in which the rows are only partially filled.

spatial authority: an attribute of a field that determines which functions within a program can refer to it or change its contents.

square brackets: the [and] symbols.

stack: a restricted linear list in which data be inserted and deleted at only one end, called the *top*.

stack memory: in C, the memory management facility that is used to store local variables while they are active.

stamp coupling: the communication technique between modules in which data are passed as a structure; often results in unrequired data being passed.

standard error stream: the text file automatically opened by C for display of error messages (*stderr*).

standard input stream: the text file automatically opened by C for input to a program (*stdin*).

standard library: any of a collection of libraries containing functions required by the C standard provided by an implementation of the C language.

standard output stream: the text file automatically opened by C for output from a program (*stdout*).

standard type: one of the intrinsic C types that are considered atomic; that is, that cannot be broken down (*void, char, int, float*).

statement: a syntactical construct in C that represents one operation in a function.

statement section: the section in a program where the executable statements are written.

***static*:** a storage class that designates (a) that a local field must maintain its contents throughout the execution of a program (contrast with *auto*), or (b) that a global variable's name must not be exported to the linker (contrast with *extern*).

static memory: memory whose use (e.g., for a variable) does not change during the running of a program.

static memory allocation: memory whose location is determined by the compiler and therefore preset before run time. Contrast with *dynamic memory allocation*.

***stderr*:** see *standard error stream.*

***stdin*:** see *standard input stream.*

***stdout*:** see *standard output stream.*

stepwise refinement: a design methodology in which a system or program is developed from the top down; starting with the most inclusive, each module is decomposed and refined until the meaning of a component is intrinsically understood.

storage class: an attribute of a field that determines its spatial and temporal usage. See also *auto, extern, register, static*.

stream: in C, the flow of data between a file and the program; a sequence of bytes in time.

string: in C, a variable-length sequence of characters delimited by a null character.

string constant: a constant comprised of a sequence characters enclosed in double quotes.

strongly connected graph: a graph in which each there is a path from every node to every other node; contrast with *weakly connected graph*.

structure: a named collection of fields grouped together for processing; see *record*.

structure chart: a design and documentation tool that represents a program as a hierarchical flow of functions.

structure variable: in C, a structure definition that cannot be used as a type.

stub: a temporary and incomplete function used to test function interfaces during program development

subscript: an ordinal number that indicates the position of an element within an array. See also *index*.

summation: an algorithm that adds a list of data items.

suffix: a modifier to a numeric constant that indicates its type when used in an expression.

switch: see *flag*.

switch statement: the C implementation of the multiway selection.

symbolic language: a computer language, one level removed from machine language, that has a mnemonic identifier for each machine instruction and has the capability of symbolic data names.

syntax: the "grammatical" rules of a language. In C, the set of keywords and formatting rules that must be followed when writing a program.

system development life cycle: a model of the steps required to develop software that begins with the need for the software and concludes with its implementation.

system development software: any computer tool used to develop software, such as but not limited to compilers, debuggers, and documentation tools.

system software: any software whose primary purpose is to support the operation of the computing environment. Contrast with *application software*.

system support software: software used for nonapplication processing, such as system utilities.

T

tagged structure: a C structure in which the structure is given an identifier that can be used to declare variables of the structure type.

tape storage: an auxiliary storage medium that stores data as a sequential file on a magnetic recording surface.

temporal authority: an attribute of a field that determines when a field is alive and active during the execution of a program.

temporal cohesion: a module design in which processes are combined because they all need to be processed in the same time sequence.

terminating condition: in a loop, the condition that stops a loop. See also *limit test expression*.

ternary expression: in C, an expression that contains three operands (the conditional expression is the only ternary expression in C).

testability: an attribute of software that measures the ease with which the software can be tested as an operational system.

testing: see *program testing, blackbox testing*, or *whitebox testing*.

text editor: software used to create and change text files, such as a word processor or a source program editor.

text file: a file in which all data are stored as characters. Contrast with *binary file*.

timeliness: an attribute of software that measures responsiveness of a system to a user's time requirements.

time-sharing environment: an operating system concept in which more than one user has access to a computer at the same time.

token: in C, a syntactical construct that represents an operation or a flag, such as the assignment token (=).

top: in a stack, the next element to be removed.

top–down design: a program design concept in which a design progresses through a decomposition of the functions beginning with the top of the structure chart and working toward the lowest modules. See also *stepwise refinement*.

transaction file: a file containing relatively transient data that is used to change the contents of a master file.

transform analysis: an analytical process that creates a program design by classifying modules as input, process, or output.

transform module: the function in a program that takes input from the afferent modules and prepares it for processing by the efferent modules.

translation unit: in C, a temporary compilation file used to store modified source code.

translator: a generic term for any of the language conversion programs. See also *assembler* and *compiler*.

traversal: an algorithmic process in which each element in a structure is processed once and only once.

true: a Boolean value used to indicate that an expression is true.

two-dimensional array: an array in which each element contains one array. See also *multidimensional array*.

two-way selection: a selection statement that is capable of evaluating only two alternatives. In C, the *if...else* statement. Contrast with *multiway selection*.

type: a set of values and a set of operations that can be applied on these values.

type qualifier: a modifier used in the definition of an object.

U

unary expression: an expression that consists of one operator and one operand.

unary minus: in C, the operator that complements the value of an expression.

unbuffered input/output: input or output that is transmitted directly to or from memory without the use of a buffer.

underflow: an event that occurs when an attempt is made to delete data from a data structure and it is empty.

Unicode: The internation character set used to define valid characters in computer science. Each character is represented using a 32-bit unsigned number.

union: C term for *variable structure*.

unsigned: a type modifier indicating that a numeric value may be only positive. Contrast with *signed*.

update: (1) in array processing, the process that changes the contents of an element; (2) in file processing, a mode that allows a file to be both read and written.

update mode: a file process mode that specifies that the file will be both read and written.

upward communication: data flow from the called function to the calling function.

user-defined function: any function written by the programmer, as opposed to a standard library function.

user prompts: monitor messages to a user that request the user input one or more values.

V

value error: an input value that does not satisfy specified criteria.

variable: a memory storage object whose value can be changed during the execution of a program. Contrast with *constant*.

variable-length array: an array in which the number of elements can be changed during the run time. Contrast with *fixed-length array*.

variable structure: a data structure in which two or more types of data may occupy the same positions within the structure. In C, a *union*.

visibility: the temporal authority for a field that designates where in the program it can be accessed or changed; see *scope*.

vertex: a node in a graph.

***void*:** the absence of data.

***void* functions:** functions whose return value is *void*.

***void* pointer:** a generic pointer type that can hold the address of any data type or structure.

***volatile*:** an attribute of a variable that indicates that it may be accessed or changed by functions beyond the control of the compiler.

W

waterfall model: a system development life cycle in which each phase of development, such as requirement development and design, is completed before the next phase starts.

weakly connected graph: a graph in which there is at least one node with no path to any other node: contrast with *strongly connected graph*.

weighted graph: a graph whose lines are weighted.

***while* loop:** a sentinel-controlled, pretest loop in C.

whitebox testing: program testing in which the internal design of the program is considered; *clear box testing*. Contrast with *blackbox testing*.

whitespace: in C, the space, vertical and horizontal tabs, newline, and form-feed characters.

width modifier: in a format string, a conversion code modifier that specifies the input maximum or output minimum size.

write mode: the attribute of a file indicating that it is opened for output only.

write state: one of three states that an open file can assume. A file in the write state can only be used for output. See also *error state, read state*.

Index

Argument Type	Size Specifier	Code
integral	hh (char), h (short), none (int), l (long), ll (long long)	i
integer	h (short), none (int), l (long). ll (long long)	d
unsigned int	hh (char), h (short), none (int), l (long), ll (long long)	u
character octal	hh (unsigned char)	o
integer hexadecimal	h (short), none (int), l (long), ll (long long)	x
real	none (float), l (long double), L (double)	f
real (scientific)	none (float), l (long double), L (double)	e
real (scientific)	none (float), l (long double), L (double)	g
real (hexadecimal)	none (float), l (long double), L (double)	a
character	none (char), l (wchar_t)	c
string	none (char string), l (wchar_t string)	s
pointer		p
integer (for count)	none (int), hh (char), h (short), l (long), ll (long long)	n
set	none (char), l (wchar_t)	[

Sizes, and Conversion Code for *scanf* Family

Argument Type	Flag	Size Specifier	code
integer	-, +, 0, space	hh (char), h (short), none (int), l (long), ll (long long)	d, i
unsigned integer	-, +, 0, space	hh (char), h (short), none (int), l (long), ll (long long)	u
integer (octal)	-, +, 0, #, space	hh (char), h (short), none (int), l (long), ll (long long)	o
integer (hex)	-, +, 0, #, space	hh (char), h (short), none (int), l (long), ll (long long)	x, X
real	-, +, 0, #, space	none (float), l (long double), L (double)	f
real (scientific)	-, +, 0, #, space	none (float), l (long double), L (double)	e, E
real (scientific)	-, +, 0, #, space	none (float), l (long double), L (double)	g, G
real (hexadecimal)	-, +, 0, #, space	none (float), l (long double), L (double)	a, A
character	-	none (char), l (w-char)	c
string	-	none (char string), l (w-char string)	s
pointer			p
integer (for count)		none (int), h (short), l (long)	n
to print %			%

Flags, Sizes, and Conversion Codes for *printf* Family

Behrouz A. Forouzan & Richard F. Gilberg

This third edition of *Computer Science: A Structured Programming Approach Using C* offers introductory students a fully updated, comprehensive survey of computer science theory and the C programming language. Now in full color, Forouzan and Gilberg continue to use a highly visual approach to explain fundamental programming concepts. Changes to the new edition include full compliance with the C99 standard, expanded sections on inter-function communication and the basic data structures, and a thoroughly revised Chapter 3.

Features

- Updated to comply with the C99 standard, including discussions of complex arithmetic and Unicode.
- Features a new full-color page design that enhances figures and presents programs as they might be shown in the compiler.
- Offers a groundbreaking visual approach, including a wide variety of figures, tables, and programs.
- Uses a large number of examples, ranging from code snippets to complete implementations requiring several functions, providing the student with a range of techniques to study and practice.
- Contains extensive end-of-chapter pedagogy, including a summary, keywords, tips, and a robust problem set featuring review questions, exercises, problems, and projects.

Related Title

Data Structures: A Pseudocode Approach with C Second Edition

Richard F. Gilberg & Behrouz A. Forouzan
0-534-39080-3

About the Authors

Behrouz A. Forouzan has more than 32 years of electronics and computer science experience in industry and academia. His industry experience includes designing electronic systems. After leaving industry, he joined De Anza College as a professor of computer science. In addition to this text, he has authored and co-authored nine other textbooks, including *Data Structures: A Pseudocode Approach with C* and *Foundations of Computer Science*.

Richard F. Gilberg has more than 40 years of computer science experience in industry and academia. His industry experience includes the development of large application systems, database administration, and system testing. After leaving industry, he joined De Anza College as a professor of computer science. In addition to this text, he has also co-authored several others, including *Data Structures: A Pseudocode Approach with C*.

COURSE TECHNOLOGY
CENGAGE Learning

For your lifelong learning solutions, visit **course.cengage.com**
Purchase any of our products at your local college store or at our preferred online store **www.ichapters.com**

ISBN-13: 978-0-534-49132-1
ISBN-10: 0-534-49132-4
90000 >
9 780534 491321