Instructor's Manual

Problem Solving and Program Design in C, 6e
Jeri Hanly & Elliot Koffman

Preface

This manual is designed to support instructors using the text *Problem Solving and Program Design in C,* Sixth Edition by Jeri R. Hanly and Elliot B. Koffman. This supplement includes chapter-by-chapter summaries and suggestions, answers to internal self-check and programming exercises, answers to the review questions for each chapter, a collection of true/false, multiple choice, and short answer questions to assist in constructing chapter tests, and solutions to selected programming projects.

I thank especially Joan C. Horvath of the Jet Propulsion Laboratory, California Institute of Technology, for contributing her solution to one of the programming projects that she designed. I am very grateful to Howard University former student Paul Onakoya and to University of Wyoming former graduate student Mark A. Thoney, who wrote or verified solutions to numerous internal exercises, review questions, and programming projects. I thank also Temple University former computer science students Lynne Doherty, Andrew Wrobel, Steve Babiak, and Donna Chrupcala, who contributed solutions to internal exercises and review questions. I acknowledge the assistance of University of Wyoming former computer science and engineering students Jim Anderson, Kenneth Bower, Amy Fletcher, Joe Fuhrman, Chris Hansen, John Regnier, Russell Updike, and Brent Youngers, who provided solutions to programming projects.

I wish to express special gratitude to Addison-Wesley Computer Science Executive Editor Michael Hirsch and to Computer Science Editorial Assistant Stephanie Sellinger for their unfailing professional support.

J.R.H.

Part I
Pedagogical Notes

Chapter 1 Overview of Computers and Programming

Chapter Objectives

1. Present a brief historical overview of the development of computers.
2. Describe the major components of a computer system and how they work together to solve problems and manipulate data.
3. Define the major categories of software and the kinds of languages in which they are implemented.
4. Introduce the steps of the software development method.
5. Explain the process of writing, compiling, and executing high-level language programs.
6. Present the use of the software development method in the solution of an elementary programming problem.

Notes and Suggestions

This chapter contains broad coverage of general information about computers, their history, major components, and the process of developing programs. The amount of time spent on the above objectives will be related to your students' previous experience with computers. If this is a first course for most of your students, then you may want to proceed more slowly. Regardless of your students' prior experience with computers, it would be advisable to augment the chapter with demonstrations of the use of the program development environment on the computer system they will be using for their programming assignments in this class.

Require students to use good programming style in their very first program and in all subsequent coding assignments. If you have a strong preference for a style different from the style used in the text, provide your students with an explicit list of guidelines to follow. The style in the text follows very closely the style rules mandated for use by developers in two major C-using corporations. Be sure that all of your sample programs follow the rules that you want your students to follow. To facilitate giving students specific recommendations for improving their programs, distribute a key listing your normal deductions for failure to meet your expectations. Using the letters from the key will save you a lot of writing. Here is a key you could use as a first draft.

		% usually deducted
A.	Missing statement of purpose	8
B.	Inadequate commenting	8
C.	Names are not meaningful	8
D.	Indentation does not indicate program structure	4
E.	Program will not compile	100
F.	Program produces incorrect results	20–100
G.	Insufficient testing (program branch never executed, borderline case not tested, prescribed test case results not submitted)	4–20
H.	Output not annotated to demonstrate correctness of results	8
I.	Program solves wrong problem	8–100
J.	Algorithm inefficient or difficult to follow	10
K.	Use of unnamed constant	4

Section 1.5 contains the first of several complete case studies that describe the development of solutions to programming problems using the software development method. It may be difficult to motivate students to work through all the steps of the software development method on the easy programs that appear in the first chapters of the text. It may be helpful to remind them that the methods discussed in the case studies are modeled after those used by modern computing professionals.

New Terms

Section 1.1

John Atanasoff	software	ENIAC
computer program	computer chip	hardware
personal computer	minicomputer	binary number
mainframe	workstation	palmtop computer
supercomputer		

Section 1.2

input device	mouse	secondary storage
printer	address	disk drive
stored program concept	file	keyboard
program file	hard copy	modem
main memory	disk	data file
byte	data retrieval	bit
central processor unit (CPU)	read-only memory(ROM)	output device
data storage	memory cell	hard disk
volatile	random access memory (RAM)	USB flash drive
output file	monitor	local area network (LAN)
wide area network (WAN)	graphical user interface	World Wide Web (WWW)
function keys	optical drive	subdirectory
fetch an instruction	icon	wireless network
computer network	file server	cursor
register	digital versatile disk (DVD)	directory
kilobyte	megabyte	gigabyte
terabyte	digital subscriber line (DSL)	cable Internet access

Section 1.3

programming language	syntax	high-level language
FORTRAN	machine language	BASIC
assembly language	COBOL	LISP
Ada	Java	C and C++
language standard	system software	application software
source file	object file	operating system
integrated development environment	compiler	syntax error
linker	word-processor	editor
program output	input data	loader
binary string		

Section 1.4

software engineer	software development method	abstraction
analysis	design	implementation
testing	problem inputs	problem outputs
subproblems	stepwise refinement	divide and conquer
top-down design	requirements specification	desk checking
maintenance		

Chapter 2 Overview of C

Chapter Objectives

1. Present the origins of the C language, and introduce the structure of simple C programs through the presentation of small sample programs.

2. Describe the simple data types int, double, and char.

3. Introduce the basic use of the input/output functions printf and scanf.

4. Present the preprocessor directives #include and #define.

5. Discuss the use of input and output format strings with placeholders for character and numeric data.

6. Introduce the proper use of comments as a means of program documentation.

7. Present the assignment statement along with the basic arithmetic operators and type casts

and their use in writing simple arithmetic expressions.

8. Describe the process of writing simple programs that perform input, computation of numeric results, and output.

9. Discuss input/output redirection and program-controlled files, and present the differences between interactive and batch processing of data.

Notes and Suggestions

Section 2.1. The beginning student often is confused about the difference between reserved words and standard identifiers, so you should emphasize the difference. Students will also benefit from a clear presentation of the two stages of the translation of a C program. If they understand the role of the preprocessor, they will see as reasonable the fact that a constant macro's value cannot be modified by a program. Emphasize the importance of choosing meaningful names for variables. If some students resist the extra typing required to enter meaningful names, you may want to prepare two versions of the same program, one with single-letter names and the other with meaningful names. Students will quickly see which is more readable.

Section 2.2. Stress the meaning of the term "data type." Understanding this term early will make it easier for students to comprehend the concept of defining their own types.

Section 2.3. Students often confuse the C assignment operator with the equal sign from algebra, which makes the statement x = x + 1; appear as nonsense. When you read an assignment statement aloud, always pronounce the = as "becomes" or "is assigned" to emphasize that the assignment statement's effect is to compute a value to be placed in the memory cell named by the variable on the left.

In a similar vein, some students incorrectly view assignment statements as spreadsheet-like identities that automatically update the left side whenever the right side changes. Consider the following C statements:

```
b = 2;
a = b + 1;
b = 3;
printf("%d", a);
```

A student who expects an output of 4 has fallen into this trap.

When presenting the input/output functions scanf and printf, emphasize the differences between the input list of scanf and the output list of printf. We have intentionally delayed presentation of the input of strings so that at this point the & operator is required on all entries in scanf's input list. You should draw a picture of computer memory for a particular input/output statement sequence, and show students the difference between what scanf needs to know (i.e., addresses of variables) and what printf needs to know (i.e., the values to print). Also do exercises that stress the fact that the numeric format placeholders %d and %lf skip over blanks and carriage returns in the input, but the %c placeholder skips nothing unless it is preceded by a blank in the format string. Another instructive example is one in which a call to printf begins a line of output, but does not complete it with the \n escape sequence. The student should be aware that the output may not be displayed if the program terminates because of an error before the line is completed.

Section 2.4. Beginning students of C, especially those who have taught themselves to program in another language, are frequently very resistant to principles of good program documentation and the importance of readability. "War stories" about the grief of maintenance programmers expected to update unreadable code or of your own former students who have encountered poor documentation in their first jobs will help to counter the notion that the only measure of a good program is whether it works. The following definition may help to put comments, meaningful names, and eye-pleasing spacing in the proper perspective: A *program* is a document, part of which executes on a computer.

Section 2.5. Stress that data types are abstractions for real-world data objects. Point out that we do not need to know the details of the representation of int, double, and char variables because we are provided with operators with which to manipulate these data types and functions with which to get and display values. Encourage students to use the step-by-step evaluation approach illustrated in Fig. 2.9 through Fig. 2.11 when they need to evaluate an expression containing more than one operator.

Section 2.6. Remind students that when outputting numeric data using a placeholder with a specific field width, the field width is automatically increased to display a number whose magnitude is too large for the field. Students are often confused when they discover that the number of decimal places specified by a placeholder is absolute: the fraction is rounded to fit the field.

Section 2.7. Before covering this section, investigate how the computer system the students use handles input/output redirection. Give students explicit instructions on how to run the batch program in Fig. 2.13. If the computer system your students use does not provide simple redirection, you can skip this example and focus on the program-controlled files of Fig. 2.14.

Section 2.8. Go over with students the precise output produced by your ANSI C compiler. Emphasize the differences between syntax errors, run-time errors, and result errors.

New Terms

Introduction

C, 1972	AT&T Bell Laboratories	Dennis Ritchie
UNIX		

Section 2.1

comment	preprocessor	preprocessor directive
#include	#define	library
functions	printf	scanf
<stdio.h>	declarations	executable statements
constant macro	standard libraries	main
prototype	reserved words	identifiers
user-defined identifier		

Section 2.2

variable	variable declaration	data type
identifier	int	void
double	char	scientific notation

Section 2.3

executable statement	assignment statement	assignment operator
expression	arithmetic operator	input operation
output operation	input/output functions	function call
format string	input list	standard input device
placeholder	conversion specification	address-of operator
return statement	print list	escape sequence
newline	prompting message	function argument

Section 2.4

blank space	program documentation	

Section 2.5

integer division	remainder operator	step-by-step evaluation
mixed-type expression	unary operator	binary operator
operator precedence rule	associativity rule	right associativity
left associativity	evaluation tree	type cast

Section 2.6

field width	right justified	decimal places

Section 2.7

interactive mode	batch mode	echo print
input/output redirection	program-controlled file	file pointer

Section 2.8

bugs	debugging	error message
syntax error	compilation error	run-time error
diagnostic message	compiler listing	logic error

Chapter 3 Top-Down Design with Functions

Chapter Objectives

1. Present programs illustrating top-down problem-solving, the divide-and-conquer strategy, and the process of stepwise refinement.
2. Discuss the use of structure charts (without data flow information) as a means of showing the hierarchical relationships among a problem's subproblems.
3. Introduce the concept of a structured program as a combination of control structures with one entry and one exit point.
4. Present the use of void functions and of functions returning a single result as a means of building modularized programs.
5. Introduce input parameters as a means of passing data into functions.
6. Describe the syntax and semantics of user-defined functions that return no result and of functions that return a single result.
7. Illustrate the notion of code reuse by discussing and calling several standard library functions used in computation.

Notes and Suggestions

General suggestions. It is important to reinforce throughout the course the systematic nature of the top-down approach to solving problems. Although it is difficult to motivate students to follow all the steps when solving the simple problems found early in the course, adherence to this approach will help students develop the programming habits necessary to tackle larger problems. You may want to have students hand in top-level algorithms prior to having them code their solutions in order to reinforce these habits.

Section 3.1. Students should begin to use the top-down approach for analyzing all problems. Point out that the solution to any programming problem is conceptually simpler if viewed hierarchically as a tree of subproblems.

Often the first step in solving a problem (understanding the problem) is the hardest. Requiring students to turn in written samples of a program's input and output prior to working on an algorithm can help them determine early if they understand a problem.

Section 3.2. Steer students away from the NIH syndrome (not invented here syndrome). Successful programmers learn to adapt existing code to new problems, create reusable code, and use off-the-shelf software. In this section, students meet examples of function input arguments in their study of selected library functions for mathematical computation. Emphasize the fact that any of these library functions can be called at any place in a program where the value the function returns is needed.

Section 3.3. Draw structure charts for problems discussed in earlier sections of the chapter to illustrate the hierarchical relationship among a problem's subproblems.

Section 3.4. New programming students often find the concept of functions difficult to understand. It is often helpful to relate the calling of functions to a person's calling in experts to handle various aspects of a difficult project. For instance, building a house requires laying a foundation, constructing the walls and roof, installing plumbing, and so on.

Emphasize that the point of defining functions is to organize the C implementation so it parallels the top-down solution of a problem. There is more to this organization than simply breaking a large program into smaller pieces; each piece must be a coherent module that performs one of the major subtasks contributing to the solution of the problem. Point out that the syntax of a function subprogram is similar to the syntax of function main: each has a heading and a body consisting of variable declarations and executable statements.

Section 3.5. Be sure that the student understands the difference in calling protocol between void functions and functions that return a value. Point out repeatedly that a calling module must do something with the result returned by a function. Otherwise, the value returned will be lost.

In the engineering of correct, readable, maintainable software, the consistent use of function interface comments is essential. From now on students should be required to include a statement of purpose on every function they write.

Students may have some confusion regarding the purpose of preconditions. Preconditions describe facts about the parameters that the function assumes to be true.

Emphasize that the function interface comment and the function prototype represent the conditions of a contract between the function and its caller. The function guarantees that its purpose will be accomplished and any postconditions satisfied as long as the caller provides the right number of arguments whose types are compatible with the function parameters, and as long as the arguments meet the stated preconditions.

Section 3.6. Omitting a necessary #include directive can result in erroneous results on some systems. Since students frequently do forget to include the math library when it is needed, you should check how your system deals with this error.

New Terms

Section 3.2

function result	reuse

Section 3.3

structure chart

Section 3.4

function definition	local variable	transfer of control
function prototype		

Section 3.5

input argument	output argument	actual argument
function data area	driver	argument list
function interface comment	formal parameter	precondition
postcondition		

Chapter 4 Selection Structures: if and switch Statements

Chapter Objectives

1. Describe the syntax and semantics of the single- and double-alternative if statements.
2. Present relational, equality, and logical operators and their use in the conditions of decision steps.
3. Introduce the syntax and semantics of compound statements.
4. Discuss the construction of nested and multiple-alternative if statements to implement complex conditional logic and decision tables.
5. Introduce selected flowcharting symbols as a means of diagramming decisions in algorithms.
6. Present the use of data flow information in structure charts.
7. Discuss a technique for hand tracing an algorithm.
8. Introduce the syntax and semantics of the switch statement as a multiple-alternative decision structure.

Notes and Suggestions

Section 4.1. Emphasize the meaning of the term "control structure," and use it when introducing each new selection or repetition statement.

Section 4.2. Stress the difference between the assignment operator = and the equality operator

==. You may want to intentionally misuse the assignment operator a few times to sensitize students to being alert for this common error. Be sure to explain how an if statement with an = in the condition will execute. The text stresses the fact that the concept "true" can be represented by any nonzero integer so that students will be able to understand the behavior of statements containing the = where the == belongs.

Students with a weak mathematics background may have difficulty with logical operators and expressions. Encourage students to take a very analytical approach to evaluating expressions that include logical operators. Have them refer to Tables 4.3 to 4.5 and use the step-by-step method of evaluation illustrated in Fig 4.1. Emphasize the importance of the programmer's checking that both operands of a logical operator are expressions representing logical concepts. Because of C's use of type int to represent logical values, compilers can rarely mark conceptual errors in logical expressions. Be sure to explain short-circuit evaluation of logical expressions even though the full implications cannot be discussed until later.

Section 4.3. Emphasize the fact that a C compiler expects to see only one statement on a branch of an if statement. Stress especially that indentation helps the reader but conveys no meaning to the compiler. Remind students that neither the condition of an if statement nor the else keyword is normally followed by a semicolon.

Section 4.4. Encourage your students to practice desk-checking (tracing) algorithms on a regular basis. When a student comes to you with a programming problem, have the student trace the program while you watch.

Section 4.5. Adding data flow information to structure charts helps prepare students for the use of input and output parameters in user-defined functions, a topic that will be presented in Chapter 6. You can relate the use of data flow information to the discussion of input arguments and function results in Section 3.8.

Section 4.6. The second water-bill case study gives you another opportunity to stress to your students the importance of code reuse in successful program development.

Section 4.7. Give students practice with converting nested if statements to multiple-alternative if statements. Also present cases where the conversion is not possible. Remind students to be careful about the use of compound statements within complex decision structures. Many commercial C users require developers to bracket all branches of a multiple-alternative decision if just one branch contains a compound statement.

Section 4.8. You should be aware that the text does not present the switch statement in its most general form. We have intentionally shown the switch only in the context where it represents an implementation of a structured multiple-alternative decision statement. Emphasize to students the critical role of the break statements within the switch.

Section 4.9. The topics of Chapter 4 include numerous constructs that students can misuse without creating a syntax error. Be sure to underline the problems that will result from poorly constructed conditions such as (0 <= x <= 4) and (x = y).

New Terms

Section 4.1

control structure	compound statement	sequence
selection	repetition	

Section 4.2

logical expression	condition	relational operator
equality operator	complement	short-circuit evaluation
DeMorgan's theorem		

Section 4.3

if statement	flowchart	false branch
empty statement	true branch	

Section 4.4

tracing an algorithm	hand trace	desk check
true task	false task	

Section 4.5

decision step	data flow information	pseudocode
input to a step	output of a step	cohesive function

Section 4.7

nested if statement	multiple-alternative decision	decision table
mutually exclusive conditions		

Section 4.8

switch statement	controlling expression	case label
break statement	default label	

Chapter 5 Repetition and Loop Statements

Chapter Objectives

1. Introduce the concept of repetition in algorithms.
2. Describe the syntax and semantics of the C while, for, and do-while statements.
3. Demonstrate the use of loops to compute sums and products of lists of numbers.
4. Discuss the use of the while and for statements in the construction of counter-controlled loops, conditional loops, sentinel-controlled loops, and endfile-controlled loops.
5. Present the use of the compound statement as a loop body.
6. Introduce the use of loops to display tables of information.
7. Describe and demonstrate the nesting of control structures.
8. Describe the use of type int variables as program flags.
9. Present several C operators that have side effects on their operands: increment, decrement, and compound arithmetic assignment operators.

Notes and Suggestions

Section 5.1. Emphasize the conceptual differences among the different kinds of loops before presenting C syntax for any of them.

Section 5.2. Explain that the loop repetition condition is tested before each loop iteration, so a loop will be executed zero times if the condition is initially false. Stress that the programmer must ensure that all variables occurring in the loop repetition condition are initialized prior to encountering the while statement during program execution.

Infinite loops can cause serious problems for the student on some microcomputer systems, because the computer may need to be reset to abort a runaway program. Find the most graceful abort procedure available on your system, and teach your students how to use it. Warn them to be sure to save a program to disk before attempting to run the program.

Section 5.3. You can play the following game to help students understand the use of memory cells in a program that accumulates the sum of a list of numbers. Tell your students that you want them to add up 12 numbers. Do not allow them to write anything down, and instruct them to call out the desired sum as soon as they know it. Then read the list of numbers slowly, using data that is very easy to add, such as small multiples of 5. When the sum is found, ask questions to help the students identify what information they had to remember at each point in the exercise. Similar games can preface the computation of the product of a list of numbers and the use of a sentinel to signal the end of a list of data.

Section 5.4. Stress the fact that the for statement, although apparently more complex than the while, is the most popular iteration statement among C programmers in industry because of its provision of consistent locations for the loop initialization step, the loop repetition

condition, and the update step. Have students hand trace numerous examples of for statements.
Emphasize that the increment and decrement operators should be used *only* when the side effect is needed. After learning about the ++ operator, many students begin to substitute ++x or x++ for the expression x + 1, even when there is no need to increment x. Also stress that the target of one of these operations cannot appear again in the same expression in which the ++ or -- is applied.

Section 5.5. Point out the essential differences between counter-controlled loops and the more general event-controlled conditional loops. Emphasize that the presence of a variable that is counting something does not necessarily imply the need for a counter-controlled loop.

Section 5.6. Loop design is often difficult, so students appreciate some guidelines to help them design loops correctly. When solving problems in class, use the questions in Table 5.4 to drive the loop design process. Emphasize the correct form of a sentinel-controlled loop and the importance of the priming scanf call. Be sure that students understand the flaws in the incorrect sentinel loop displayed in the text. The section entitled "Infinite Loops on Faulty Data" is critical. Do several in-class similar to the text's example.

Section 5.7. Students who have not mastered the skill of tracing programs will find the task of tracing nested loops overwhelming. Review the process of hand tracing, and give students plenty of practice.

Section 5.8. Stress the fact that the availability of a third looping construct makes it critical that the programmer carefully evaluate each application of iteration. If there is *any* possibility that a loop body should be skipped altogether, the programmer must not use the do-while. Be sure that students understand the flaws in the do-while version of the sentinel-controlled loop of Self-Check Exercise 1.
Some students have difficulty understanding the need for program flags. Perhaps the problem is their failing to see that various parts of a program must communicate with one another. After meeting flags, many students will be tempted to write (flag == TRUE). Emphasize that a type int variable designated as a logical flag is itself a perfectly valid condition to use in an if statement or as a loop repetition condition.
C programs frequently need to discard extra characters from an input line. Check that students understand how this processing is accomplished in Fig. 5.13.

Section 5.9. Students should practice converting while loops to for loops and converting for loops to while loops. Verify that the statements they choose for initialization and update of for loops actually affect the loop control variable. Give an example of a loop that works but misleads the reader by placing statements unrelated to the loop control variable in the initialization and update slots of the for construct.

Section 5.10. The insertion of extra calls to printf is an important debugging aid in an environment that does not provide an on-line debugger. Stress the importance of including the \n at the end of the format string of a debugging output. If an on-line debugging program is available, take the time to present documentation on its use to your students.

New Terms

Introduction

loop

Section 5.1

loop body

Section 5.2

loop control variable	counter-controlled loop	counting loop
while statement	initialization step	iteration
pass	loop repetition condition	update step
exiting a loop	infinite loop	

Section 5.3

accumulator	compound assignment operator

Section 5.4

for statement	initial value	final value

increment operator	decrement operator	prefix
postfix	side effect	

Section 5.5

conditional loop

Section 5.6

loop exit condition	sentinel-controlled loop	sentinel value
endfile-controlled loop	zero iterations	EOF value

Section 5.7

nested loops

Section 5.8

do-while statement	flag	flag-controlled loop
input validation		

Section 5.10

debugger program	off-by-one error	loop boundaries
single-step execution	breakpoint	diagnostic printf

Chapter 6 Modular Programming

Chapter Objectives

1. Present the syntax and semantics of pointers used as function output parameters and as input/output parameters.

2. Discuss C's scope rules governing the visibility of constant macros, function names, formal parameters, and local variables.

3. Introduce the relationship between data flow shown on structure charts and the process of program design through stepwise refinement.

4. Present the use of the function interface comment and the function prototype as documentation of all aspects of the function needed by the caller of the function.

Notes and Suggestions

General Suggestions. This is one of the most difficult chapters in the first half of the text. Take extra time to discuss the concepts and have students work many examples.

Section 6.1. This section is the first to present the concept of a pointer in C. Resist the temptation to introduce pointers in general. The concept of an output parameter is an important aspect of a disciplined approach to program design. Give your students plenty of practice in defining and using output parameters. Oral exercises using diagrams like those in Fig. 6.4 and in Self-check Exercise 3 can be very helpful in identifying situations where students are confused about the need for or meaning of the address-of and indirection operators.

Section 6.2. Emphasize that a function can communicate with the caller in three ways: receiving information through input parameters, transmitting information through the function result and through output parameters, and updating data through input/output parameters. Be sure that students understand that any parameter involved in transmission or update of data must be declared as a pointer.
Table 6.3 summarizes the kinds of functions presented in the chapter. You may want to review function examples from chapters 3-6 and ask your students to place each in one of the categories shown in Table 6.3.

Section 6.3. Understanding the concept of scope is a real challenge. Students often have difficulty with test questions requiring them to trace the execution of programs that have local variables, function parameters, and function names with naming conflicts. Students will need to see several such examples presented in class if they are to do well on this type of exercise.

Section 6.4. Advise your students to draw diagrams like the one in Fig. 6.10 whenever they are manipulating output parameters. Only when they have a clear visual image of how pointers are used will they be certain of where to apply the & and * operators. Also encourage students to use Table 6.5 to help them analyze the roles of parameters in functions.

Section 6.5. Starting with function main, show your students how a well-designed program allows the reader to choose the level of detail to explore. Discuss the degree to which the reader can understand the entire system by reading just the main function.

Section 6.6. The use of stub functions is sometimes difficult to motivate when dealing with simple programs. Students find it easier to comprehend their usefulness after working on development of a program as part of a group.

New Terms

Section 6.1

input parameter	output parameter	address
pointer	side effect	indirection operator
indirect reference	direct reference	

Section 6.2

input/output parameter	sort

Section 6.3

scope	visible

Section 6.5

stub

Section 6.6

top-down testing	unit test	bottom-up testing
system integration test		

Chapter 7 Simple Data Types

Chapter Objectives

1. Discuss the internal representations of type int, double, and char values.
2. Introduce the problem of numerical inaccuracies in computations using type double values.
3. Review automatic and explicit conversion of data types.
4. Present the syntax and semantics of enumerated types.
5. Introduce an algorithm for finding approximations of function roots.

Notes and Suggestions

Section 7.1. This section provides a brief introduction to the topics of internal representation of numbers and computational error. Students may be surprised to learn that computers do not perform real arithmetic calculations precisely. This is a very important point to emphasize now so that students will understand the benefits of using integers when possible. This text's use of type double rather than type float postpones students' recognition of some of the most obvious consequences of representational error.

Section 7.2. Refer students to the character codes in Appendix A when discussing this section. Be sure to point out the gaps in the EBCDIC codes for alphabetic letters.

Section 7.3. Emphasize again the meaning of the term "data type." Point out that enumerated types are most commonly used to represent concepts internal to a program, since I/O of enumerated values handles only the underlying integers. Programming Project 8 is an especially good one for Computer Science majors.

Section 7.4. This section presents an algorithm for approximating roots of a function that requires only a high school algebra background. We present only one of the several syntax options for a function parameter.

New Terms

Section 7.1

round-off error	mantissa	exponent
representational error	cancellation error	arithmetic underflow
arithmetic overflow	type conversion	

Section 7.2

ASCII	printable character	control character
collating sequence		

Section 7.3

enumerated type	enumeration constant

Section 7.4

iterative approximation	root (zero of a function)	function parameter

Chapter 8 Arrays

Chapter Objectives

1. Introduce the concept, declaration, and use of one-dimensional and multidimensional arrays with elements of type int, char, or double.
2. Discuss the use of subscripted variables to manipulate array elements.
3. Describe how to pass single array elements and whole arrays as actual arguments of functions.
4. Present the use of the indexed for loop to access array elements sequentially.
5. Compare sequential and random access of array elements.
6. Discuss the representation of an array as a pointer.
7. Introduce the declaration and use of arrays as function input and output parameters.
8. Describe algorithms for searching and sorting arrays.

Notes and Suggestions

General Suggestions. The concept of a data structure consisting of many memory cells each accessed by number is a difficult one for many beginning students. Discuss the material slowly and carefully, using in-class oral exercises to help you pinpoint where your students are confused.

Section 8.1. A natural way to demonstrate the need for arrays is to ask students to try to solve problems that require storage of long lists of data. For example, you could ask your students to find the median of a large collection of exam scores.
Emphasize that an array declaration is a convenient means of allocating a large number of memory cells of the same data type. Point out that each memory cell is referenced using the array identifier and a subscript. Mention that since the subscript may be a variable or other expression, array elements can be manipulated using a repetition statement in which the subscript varies with each pass.
It is important that students be required to employ good programming style in declaring array types. Specifically, encourage them to name the array's declared size using a constant macro.

Section 8.2. Be sure that students understand the difference between an array element's subscript and the array element's value before going on to the next section. Give them lots of practice with statements like the ones in Table 8.2.

Section 8.3. It is critical that students really understand this section before attempting more complex array manipulation. Have them write many program segments that use indexed for loops for various types of array handling. Also insist that they hand trace programs that access arrays using such loops.

Section 8.4. Point out that an individual array element is handled exactly like a simple variable of the same data type. Be sure that students understand that the formal parameter corresponding to an actual argument that is an array element is a simple variable of an appropriate data type.
Students probably will have little difficulty with this section when they first study it. Only after they see how whole arrays are passed as output arguments will they begin to experience some confusion. Plan to review Section 8.4 after presenting Sections 8.5 and 8.6.

Section 8.5. Encourage students to draw diagrams of function data areas like the one in Fig. 8.10 until they understand the mechanism C uses to pass an array argument. Be sure that students realize that a function's array parameter accesses the original array passed, not a copy of its values. Insist that students use the const qualifier when declaring array input parameters to improve both the readability and robustness of their code. Also explain carefully the reasons why the declarations

```
      int list[]        and               int *list
```

are equivalent.
The fact that passing an array output argument does not require application of the address-of operator to the array name will cause problems for students. Once they learn that some output arguments don't require the &, they quickly forget to apply the & to simple output arguments as well. It is critical that students realize that the different treatment of an array output argument results from the fact that the value of an array name (with no accompanying subscript) is the *address* of the first array element.
Stress that functions returning an array result should require the caller to provide an output argument in which to store this result. Verify that students realize that dealing with an array result is different from returning a simple result from a function.

Section 8.6. Encourage students to hand trace the search and sort algorithms to gain experience with array manipulation. Also discuss the fact that searching or sorting an array of a different data type would require writing different functions. Stress that C does not convert an array of, for example, integers to an array of doubles as a result of a function call with an actual argument - formal parameter type mismatch.

Section 8.7. Emphasize the fact that declarations of multidimensional array parameters must use sizes that are constants.

New Terms

Introduction

data structure	array

Section 8.1

array element	subscripted variable	array subscript
parallel arrays		

Section 8.2

array reference

Section 8.5

const qualifier	pop	push
partially filled array		

Section 8.6

linear search	target	selection sort

Section 8.7

multidimensional array

Section 8.8

sequential access	random access

Section 8.9

subscript-range error

Chapter 9 Strings

Chapter Objectives

1. Introduce the concept of the string data type and C's implementation of varying-length strings as one-dimensional arrays of char with a special character marking the end of each string.
2. Present basic string operations such as string I/O, string assignment, substring extraction, concatenation, string length, string comparison, and type conversion between strings and numbers.
3. Discuss the use of strings as function parameters, thereby helping students to gain additional understanding of one-dimensional array parameters.
4. Describe how the pointer to a string or to any array can be returned as a function result.
5. Introduce C's library functions for character I/O, classification, and conversion.
6. Describe how C's representation of arrays as pointers makes it possible to maintain many orderings of a list of strings while only storing the strings once.
7. Prepare the student for judicious use of C string library functions by teaching them to expect to provide string functions with space in which to build function results and by sensitizing them to the critical importance of the size of the space provided.

Notes and Suggestions

General Suggestions. A major theme of this chapter is the development of the student's awareness of where each string is stored. An awareness of the relationship between the size of the array allocated for a string and the length of the string actually stored is critical if a student is to avoid the painful errors that tend to crop up with string use.

Section 9.1. Stress the importance of the null character that ends each string. Also be sure students understand that when scanf is scanning a string after seeing a %s placeholder, it does *no* checking to see if the target variable is big enough to hold the string scanned.

Section 9.2. Emphasize again the problems that can arise when using unlimited copy functions such as strcpy and strcat if there is inadequate space in the target array. Encourage students to use the limited versions (strncpy and strncat) is situations where overflow is a possibility.

Section 9.3. Continue to stress the fact that the caller of a string function must provide space into which the function can store its result. Be sure students understand the difference between strings and individual characters. However, also try to help students see strings as abstract concepts to be manipulated as units. Be sure they are not so bogged down in the details of the use of strcat that they miss out on the concept of concatenation as a useful string operation.

Section 9.4. Remind students frequently of the inapplicability to strings of the assignment operator, the equality operators, and the relational operators. You may want to have them create a library of readable comparison string functions by writing their own str_greater_than, str_equal, and str_less_than operators. In this section, just be sure they manipulate all uppercase or all lowercase letters. They will learn to deal with mixed cases in Section 9.6.

Section 9.5. Arrays of pointers are very popular among C users. This section is a gentle introduction to their use. The text's only subsequent use of arrays of pointers is for representation of arrays of string constants.

Section 9.6. This section uses the terms "facilities" and "routines" when referring to the character classifiers and converters from the ctype library because many versions of C implement these as macros.

Section 9.7. One of C's strong points is the fact that it is much easier to verify data input formats in C than in most other languages. This section presents the conversion functions that complete the collection of functions that make it possible to get a line of data as a string, use string processing to verify the data format, and then call the string version of scanf's conversion routines to store the validated values in appropriate variables.

Section 9.8. The text editor case study calls for development of several string functions that students may want to reuse in other string manipulating programs. These include delete, insert, and pos.

Section 9.9. Stress the fact that the proper behavior of the string library functions is entirely dependent on the availability of adequate space in the output argument even though the functions do not require the user to provide this size as an input argument. Be sure students really understand the error demonstrated in Fig. 9.22. Without this understanding, the reason for string output parameters in functions returning new strings will be unclear.

New Terms

Section 9.1

string constant	null character	left justification

Section 9.2

string length	string overflow	substring
token		

Section 9.3

concatenation

Section 9.4

string comparison

Section 9.5

array of pointers

Section 9.6

character classification	character conversion

Section 9.7

string-to-number conversion	number-to-string conversion	input validation

Chapter 10 Recursion

Chapter Objectives

1. Introduce the concept and usual form of a recursive algorithm.
2. Describe the process of writing recursive functions.
3. Discuss recursive algorithms for mathematical functions, for string building, and for array manipulation.
4. Discuss recursive algorithms for solving the Towers of Hanoi problem and for implementing set operations.

Notes and Suggestions

General suggestions. Recursion is an extremely important tool in computer science. Its use pervades many areas of the field, and students must become familiar with the concept early in their careers. Many students require much practice writing recursive algorithms before they really feel comfortable applying the concept of recursion. C's powerful string facilities provide us with the opportunity to use string-building examples that give the student a glimpse of the value of recursion in list processing.

Section 10.1. Emphasize the thought process illustrated in Fig. 10.3 by using similar phrases when developing any recursive algorithms in class. For example, when presenting the algorithm of the recursive function multiply, describe the recursive step as, "If I could just get someone to multiply *m* times *n*-1, then *m* times *n* would just be that number plus the value of *m*."

Section 10.2. Students must learn to trace the execution of recursive functions in order to believe that recursion really works. Provide several opportunities for students to practice drawing activation frames. Emphasize how local variables and parameters are stacked, and show the use of tracing printfs in debugging of recursive functions. However, discourage the use of a multi-level trace as an algorithm development technique. Instead, stress that during algorithm development every recursive call should be trusted to return a correct result. The student should predict the correct result of the simpler version of the problem and should check that it is properly used in creating the correct result for the top-level problem.

Section 10.3. Recursive mathematical functions are among the simplest examples of recursion. If students are familiar with inductive proofs, be sure to point out the relationship between recursive algorithms and induction.

Section 10.4. Using recursion to build assorted varying-length lists creates some of the very best examples for helping students visualize recursive algorithms. Review the meaning of sprintf as a separate topic so students will not be lost in the details of the implementations.
Recursive manipulation of numerical arrays is a challenging topic. Continue to emphasize the thought process introduced in Section 10.1 throughout the discussion of the recursive selection sort.

Section 10.5. The case study on sets is a natural bridge to the discussion of data abstraction that appears in Chapter 11. It also provides good motivation for extending students' understanding of C's representation of arrays as pointers.

Section 10.6. The Towers of Hanoi is a favorite introduction to the power of recursion for many people. Although its recursive solution is very simple, nonrecursive solutions can be extremely complex.

New Terms

Section 10.1

recursion	simple case	recursive step

Section 10.2

activation frame	terminating condition	parameter stack

Section 10.3

factorial	Fibonacci sequence	greatest common divisor

Section 10.5		*Section 10.6*
sets	string argument addresses	Towers of Hanoi problem

Chapter 11 Structure and Union Types

Chapter Objectives

1. Describe the syntax and semantics of the structure type.
2. Discuss the creation of an abstract data type through combination of a user-defined structure type and operators for manipulating the new type.
3. Contrast the use of parallel arrays and arrays of structured elements.
4. (Optional) Describe the syntax and semantics of union types.

Notes and Suggestions

General suggestions. The material in this chapter is straightforward and is usually grasped fairly easily by students. Point out the advantages of using structure types in grouping logically related data to model real-world data organization and to allow the system developer to think about problems at a higher level of abstraction.

Section 11.1. Emphasize the two essential differences between arrays and structures. Array elements must be of the same type, while structure components can be of different types. Array elements may be referenced by numeric variable indexes, but structure components must be referenced by their component names. Point out that the only built-in operations provided for complete structures are assignment and parameter passing. The built-in I/O functions can only be applied to individual structure components.

Section 11.2. Stress the fact that C does not handle all its data structures in the same way. Whereas the name of an array means the address of the first element, the name of a structure means the collection of values stored in the structure. It is helpful if students learn to view structure-type values as they view values of basic built-in types. They must see array manipulation as a separate issue.

Section 11.3. Emphasize that structure type function results are handled exactly like function results of built-in types.

Section 11.4. Software engineers realize the importance of data abstraction and information hiding in the development of large software systems. Point out that when a program performs complex number operations using only the operations that are part of the abstract data type, the ADT's implementation can change without requiring modification of programs using it.

Section 11.5. Students should work the self-check exercises carefully, since some students have difficulty determining the correct syntax for referencing components of structures that are array elements.

Section 11.6. This section can be skipped if the length of your term does not allow treatment of all of the text's topics. There is no reference to union types in later chapters.

New Terms

Section 11.1

database	structure type

component	hierarchical structure
structure tag	direct component selection operator

Section 11.2

indirect component selection operator

Section 11.4

abstract data type (ADT)

Section 11.6

union type

Chapter 12 Text and Binary File Processing

Chapter Objectives

1. Review the concept of batch processing and the use of standard input/output files as well as program-controlled text files.
2. Review the differences in C's handling of <newline> and <eof>.
3. Introduce the concept, advantages, and disadvantages of binary files.
4. Discuss functions for opening, closing, reading from, and writing to text and binary files.
5. Present the concept of a database stored as a binary file.

Notes and Suggestions

Section 12.1. Many students have trouble with the concept that the <newline> is processed as a character. Emphasize the value of seeing the standard input and output files as continuous streams of characters.

Additionally, at first students may have difficulty determining when to use a file's name and when to use the file pointer. They may also assume that the NULL pointer is the same as the null character. The fact that C's standard library functions for file pointer manipulation are inconsistent as to the position of the file pointer argument is also a factor in file pointer misuse.

Section 12.2. In this section, concentrate on comparing the new concepts and operations associated with binary files to the analogous concepts and operations for text files. Many students will be confused about the differences between text files containing numeric character strings and binary files with contents of type int or double. It is important to take the time to explain the differences.

Section 12.3. Although this section's case study is not true "database" programming, it does illustrate the database concept. The case study also provides a good introduction to the concept of sequential file search.

New Terms

Section 12.1

null pointer	end-of-file character	input stream
output stream	stdin	stdout
stderr	escape sequence	

Section 12.2

binary file	sizeof

Section 12.3

file of records	database	search parameters
record matching		

Chapter 13 Programming in the Large

Chapter Objectives

1. Introduce the concepts of procedural and data abstraction.
2. Present C's facilities for creation and use of personal libraries in the context of the use of libraries as tools for encapsulating a data object and applicable operators.
3. Explain the syntax, advantages, and disadvantages of macros with parameters.
4. Introduce storage classes extern, auto, register, static, and typedef.
5. Discuss the differences between functions developed for a single project and functions to be preserved in a library for reuse.
6. Present C's preprocessor directives for conditional compilation, and demonstrate their use in debugging and in the definition of personal libraries.
7. Explain the two optional arguments to function main.

Notes and Suggestions

Section 13.1. Remind students of examples of procedural and data abstraction seen in earlier chapters: the planet and complex data types and their associated operators in Chapter 11, the string operators of Chapter 9, and the set operators of Chapter 10.

Section 13.2. Investigate how the C implementation used by the students handles personal libraries. On UNIX- and VMS-based systems, simply storing the header and implementation files in the same subdirectory and modifying the command lines for executing the compiler and linker will incorporate personal libraries. In integrated development environments you may need to create a separate entity (sometimes called a "project") in order to use these libraries. Be sure students understand the value of the header file both in terms of the information it provides for the compiler and also in terms of the interface information it gives the programmer.

Section 13.3. Encourage students to practice building personal libraries of functions that can be reused in future projects. Discourage the "throwaway program" mind set. Be sure students understand the difference between the public view of a library represented by the header file, and the private view seen in the implementation details.

Section 13.4. Share with students some of your own war stories regarding the danger of using global variables. For practice with the meanings of the various storage classes, have students turn to earlier case studies and identify the storage classes used. Also discuss situations where the use of register or static might have been advantageous.

Section 13.5. Be sure students realize that calling the exit function is an action contrary to typical C error handling.

Section 13.6. Stress the advantages of using conditional compilation to create library implementation files that protect themselves from duplicate inclusion.

Section 13.7. You may want to prepare a version of an earlier C assignment in which values previously entered interactively are supplied on the command line as arguments to function main.

Section 13.8. Emphasize all the "warning labels" associated with the use of macros with parameters. Be sure that students understand the need for profuse parenthesis usage in macro bodies and that they see why expressions containing operators with side effects cannot be used as actual arguments in macro references.

New Terms

Section 13.1

procedural abstraction	data abstraction	logical view of object
physical view of object	information hiding	reusable code
encapsulate		

Section 13.3

implementation file

Section 13.4

extern	auto	register
static	defining declaration	global variable

Section 13.5

premature exit

Section 13.6

conditional compilation

Section 13.7

command line options

Section 13.8

macro expansion

Chapter 14 Dynamic Data Structures

Chapter Objectives

1. Review previously studied uses of pointers in C.
2. Introduce the concept of the heap and C's library functions for dynamic memory allocation and deallocation.
3. Present self-referent structure types.
4. Describe data structures that grow and shrink during program execution, such as linked lists and trees.
5. Introduce iterative and recursive algorithms for linked list creation and manipulation.
6. Introduce linked-list implementations of stacks, queues, and ordered lists.
7. Present creation and traversal of binary search trees.

Notes and Suggestions

Section 14.1. Present in class an example that demonstrates all previously studied uses of pointers in C. For example, you could extend the program in Fig. 14.2 to include an array of integers filled from a data file. You could call long_division repeatedly, using each array element as a dividend.

Section 14.2. Show students how the stack typically grows from one end of available memory while the heap grows from the other end. Point out that a program can run out of space for the stack (leading to a stack overflow message) either because the stack has grown too much or because the

heap has become too big.
Encourage students to draw diagrams of dynamically allocated space used by their programs. Show them how garbage and dangling references are created, and insist that they avoid these errors.

Section 14.3. Point out the trade-offs involved in linked list use: the use of extra space for the links in exchange for the ease of insertion and deletion and for the flexibility of having no fixed maximum list size; the extra execution time expended for dynamic memory allocation as compared to the compile-time allocation of fixed-size arrays.

Section 14.4. Give students plenty of opportunities to increase their comfort level with both linked lists and recursion by giving them numerous in-class exercises that manipulate linked lists of integers. For example, have them find (recursively) the largest value in the list, the sum of the list elements, and the product of the elements. They can also write all the list elements greater than a certain cut-off. This is also a good place to review functions as parameters: have students write a function that creates a new list containing the results of applying a function to each element of a list of numbers.

Section 14.5. Encourage students to trace the behavior of pop and push by drawing diagrams of a stack_t structure and stack_node_t nodes. Be sure they understand why both pop and push require addresses of stack structures.
Demonstrate how to use a stack in writing a nonrecursive version of the reverse_input_words function of Example 10.3.

Section 14.6. Ask students to discuss the pros and cons of including the size of a dynamic data structure in the structure type (as is done in the queue example of Fig. 14.27, and is not done in the stack of Fig. 14.23). Be sure they recognize the speed/space trade-off involved.

Section 14.7. It is worthwhile to take class time to develop both iterative and recursive versions of the function to delete a node from an ordered list. Many students will be surprised to find the recursive version easier to understand than the iterative version.

Section 14.8. To further enhance your students' appreciation of the power of recursion, after presenting the algorithms of this section, ask students to write an iterative version of the tree_search or tree_display algorithm.

New Terms

Introduction

dynamic data structure	node	

Section 14.2

malloc	heap	stack
calloc	free	dynamic memory allocation

Section 14.3

linked list	structure tag	empty list
list head		

Section 14.4

traversing a list	tail recursion	

Section 14.5

LIFO

Section 14.6

queue	FIFO	

Section 14.7

ordered list

Section 14.8

binary tree	leaf node	root node
left subtree	right subtree	parent
child	sibling	descendant
binary search tree		

Chapter 15 Multiprocessing Using Processes and Threads

Chapter Objectives

1. Introduce the students to multi-tasking.
2. Compare and contrast linear programming and parallel programming.
3. Explain linear, pseudo-parallel and parallel processing concepts.
4. Explain pre-emptive multi-tasking and context switching.
5. Introduce the students to concurrent programming methods.
6. Explain processes and multi-process programming concepts.
7. Introduce the students to multi-process programming techniques.
8. Explain interprocess communications using pipes, standard input and standard output.
9. Illustrate multi-process programming.
10. Explain multi-threaded programming concepts.
11. Introduce the students to multi-threaded programming techniques.
12. Explain thread synchronization using mutual exclusion locks.
13. Explain thread deadlocking.
14. Illustrate mutli-threaded programming.

Notes and Suggestions

Section 15.1. Use Fig. 15.1 to illustrate the differences between linear, pseudo-parallel and true parallel processing. Illustrate the parallel nature of information processing using the Read Mail / Answer Phone / Answer Door example and 5 senses analogy to explain that pseudo and true parallel processing allow computers and software to model this type of information. Be sure that the students understand that a single CPU computer, although it appears to be running more than one program at a time, is executing one program instruction at a time and time-sharing the CPU among many programs giving the appearance of parallel execution. Emphasize that without this capability a single CPU could run one program at a time.

Section 15.2. Emphasize that a program consists of a set of instructions stored in a file that is read into memory and executed and that each unique instance of an executing program is called a process. Make sure that the students understand that the operating system maintains a set of information that fully describes the state of each process in order to swap processes in and out during context switching. Use example 15.2 to illustrate how a single set of program instructions can start as one process and create another process. Be sure that the students understand that both the parent and child processes continue executing at the same program instruction after the call to the fork function and the child process has been created. Point out to the students that parent and child processes are independent of each other and as a result the parent and child processes can end independently of each another. The wait function can be used within a parent process to wait until some or all of its child processes have ended in order to synchronize this activity.

Section 15.3. Emphasize to the students that interprocess communications and executing another program from within a process are what make multi-process programming useful. Note that pipes can be thought of as files in terms of how you read and write information but that there are many other types of interprocess communications available, such as semaphores, signals, sockets, etc. in addition to pipes. Make sure the students understand that when a process executes another program that all of the information from the original process is replaced including the program instructions and memory space. As a result, locally declared variables such as pipes are no longer available and must be duplicated onto either standard input or standard output in order to be used once the process has replaced itself. Figures 15.5 and 15.6 are a complete example of how to create a child process, launch a new program from the child process and communicate between the parent and child processes using the pipe in the parent process and the standard input in the child process.

Section 15.4. Emphasize that threads are based on the idea that cooperating processes can communicate more efficiently by sharing a common memory space in order to exchange data and that thread context switching is more efficient than process context switching as a result. Make sure the students understand that only one thread, called the thread of control, is executing at a time within a process and that the threads within a process share the CPU between themselves as well as the other process that are running. Emphasize that each thread gets its own copy of the start routine specified in the pthread_create function call and that all variables local to the start routine are local to the thread. Make sure the students understand that global variables are shared among all threads and that access to these global variables must be synchronized with the mutual exclusion lock mechanism in order to prevent data inconsistency problems. Use Table 15.4 to illustrate how data inconsistency problems can occur and Table 15.5 to illustrate how data inconsistency problems can be prevented by using the mutual exclusion lock mechanism. Make sure the students understand how to use the mutual exclusion lock mechanism to prevent thread deadlocks. Figures 15.8 is a complete example of how to create a thread and update a shared resource from within two different threads using the mutual exclusion lock mechanism.

New Terms

Introduction

multi-tasking

Section 15.1

linear programming	linear processing	parallel processing
pseudo-parallel processing	time sharing	pre-emptive multi-tasking
time slice	context switch	concurrent programming

Section 15.2

parent process	child process	fork function
wait function	getpid function	execl function
WEXITSTATUS macro	zombie process	defunct process

Section 15.3

interprocess communications	pipe	half-duplex pipe
full-duplex pipe	pipe function	dup2 function
sleep function	STDIN_FILENO	STDOUT_FILENO

Section 15.4

thread	thread of control	multi-threaded
pthread_create function	pthread_mutex_init function	pthread_join function
pthread_mutex_lock function	pthread_mutex_unlock function	thread synchronization
data inconsistency	mutual exclusion locking	mutex
thread deadlock		

Chapter 16 On to C++

Chapter Objectives

1. Assist students in making the transition from programming in C to programming in its object-oriented cousin, C++.

2. Present the >> and << operators used for standard I/O in C++.
3. Describe output stream manipulators that permit formatted output in C++.
4. Introduce the concept of a reference parameter and contrast it with a value parameter.
5. Demonstrate that all of C's control structures are also available in C++.
6. Explain the major features of C++ that support object-oriented programming—class definition and overloading of functions and operators.
15. Distinguish between private and public members of a class.
16. Explain the rationale for designating a function or operator a friend of a class.
17. Present the use of header and implementation files in object-oriented programming.

Notes and Suggestions

Section 16.1. Use Fig. 16.1 as an example of how much of C++ is identical to C. Then focus on the differences: operators for I/O, output manipulators for formatting, reference parameters in functions.

Section 16.2. Emphasize that C++ allows us to design objects that act as independent agents, objects in which we can separate the object's identity and purpose from the details of how it is represented and how it behaves. The private data members are the details of the object's identity. Even though many member functions are designated "public," only their interfaces are truly public—the object's clients do not need to know the details of how the functions accomplish their purposes. Point out to your students how hiding implementation details impacts the task of updating object definitions to accommodate changes in hardware and operating systems and to incorporate new and better algorithms to improve the performance of existing code.

Be sure that students understand that they have been using overloaded operators ever since they first used "+" to add two integers in one statement and again to add two floating-point numbers in another. What is new is the fact that they are being given the opportunity to extend the overloading of operators themselves.

New Terms

Introduction

object-oriented programming (OOP)	polymorphism	

Section 16.1

output stream	insertion operator (<<)	input stream
extraction operator (>>)	value parameter	reference parameter

Section 16.2

class	operator/function overloading	signature
constructor	friend	access specifier
data member	private	public
member function	member operator	

Part II
Answers to Internal Exercises

Chapter 1 Overview of Computers and Programming

Section 1.1

2. Large real-time processing systems.

Section 1.2

2. 3 bits can have 8 values: 000, 001, 010, 011, 100, 101, 110, 111. 4 bits can have 16 values: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111.

Section 1.3

2. a. A program written in a high-level language like C is portable, unlike a program written in machine language, which can only be used on only one type of computer.
   b. A program written in a high-level language like C is quite readable compared to a machine language program, which is just a bunch of binary numbers.

4. The source program contains the lines of the program the user entered. Once the user finishes entering the program, an object program is created. The object program, created by the compiler, is a syntax free machine language translation of the source program entered. The linker combines additional object programs with the original object program to form the executable program. The final step involves the loader loading the executable file into memory for execution.

Section 1.4

2. The algorithm is created in the design stage, while the problem inputs and outputs are identified in the analysis stage.

Section 1.5

2.
```
Problem input
      quarts      -- Volume to convert.
Problem output
      liters      -- Equivalent metric volume.
Relevant formula
      liters =  0.9434 * quarts
```

Chapter 2 Overview of C

Section 2.1

2. E should be defined as a constant macro because its value should not change during program execution. If for some reason the value would need to be globally changed, using a constant macro confines this change to one place.

4. You shouldn't use a standard identifier as the name of a memory cell because by doing so you lose the ability to use that library function in your program. No, you cannot use a reserved word instead.

Section 2.2

2.

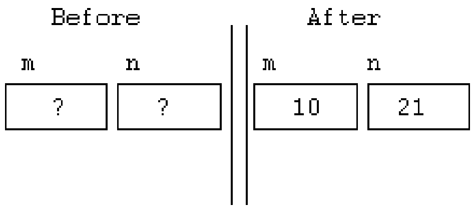
Constants	Valid/Invalid	Data Type
'PQR'	invalid	
15E-2	valid	double
35	valid	int
'h'	valid	char
-37.491	valid	double
.912	valid	double
4,719	invalid	
'true'	invalid	
"T"	not one of the listed types	
&	invalid	
4.5e3	valid	double
'$'	valid	char

Programming

1.
```
#include <stdio.h>
#define PI 3.14159

int
main(void)
{
  double  radius, area, circumf;
  int     num_circ;
  char    circ_name;
  /* executable statements omitted */
}
```

Section 2.3

2.



4. One option is to delete the third line,
```
printf("\n");
```
and insert a space after the period on the second line,
```
printf("Jane Doe. ");
```

Programming

1.
```
int first, second, third;

printf("\nEnter three integers> ");
scanf("%d%d%d", &first, &second, &third);
```

2.
```
a.    printf("The value of n is %d.\n", n);
b.    printf("The area of a square whose side length is %f cm is %f\n square cm.\n",
                side, area);
```

3.
```
#include <stdio.h>
#define PI 3.14159

int
main(void)
{
   double radius, area;

   /* Prompt for user input and get the radius. */
   printf("Enter the radius> ");
   scanf("%lf", &radius);

   /* Compute the area. */
   area = PI * radius * radius;

   /* Display the results. */
   printf("\nThe area is %f\n", area);

   return (0);
}
```

Section 2.4

2.
```
/*
 * Calculate and display the difference of two input values
 */

#include <stdio.h>
```

```
int
main(void)
{
   int first_num,         /* first input value */
   second_num,            /* second input value */
   sum;                   /* sum of inputs */

   scanf("%d%d", &first_num, &second_num);
   sum = first_num + second_num;
   printf("%d + %d = %d\n", first_num, second_num, sum);

   return (0);
}
```

The first statement scans two integers, first_num and second_num. The second statement sums those two integers. The third statement displays the first number, a plus sign, the second number, an equal sign, and the sum of the first and second numbers.

Programming

1.
```
#include <stdio.h>

int
main(void)
{
   char let;
   double num;

   /* Prompt for user input and scan. */
   printf("\nEnter a character> ");
   scanf("%c", &let);
   printf("\nEnter a number> ");
   scanf("%lf", &num);

   /* Display the results. */
   printf("\nThe character is '%c'.\nThe number is %.2f.\n", let, num);

   return (0);
}
```

Section 2.5

2.
```
1.8   *   celsius   +   32.0
            38.1
      68.58
                  100.58

(salary    -    5000.00)    *    0.20    +    1425.00
38450.00
       33450.00
                       6690.00
                                       8115.00
```

4.
```
a. 1        j. 1
b. ??       k. 3.5
c. 3        l. 21
d. 6.28318  m. 1.570795
e. ??       n. 0.0
f. 2.0      o. 3
g. 1.0      p. undefined
h. undefined      q. 7
i. ??       r. 2.3333333…
(?? means the result varies)
```

6.
```
a. a = a % c
b. x = (3 * a) / (b * c)
c. j = 4 * (i + k);
```

Programming

1. `q = (k * A * (T1 - T2)) / L;`

2. One option:
```
#include <stdio.h>

#define LETTER 'A'
```

```
      int
      main(void)
      {
         char letter1, letter2;
         int val1;
         double val2;

         letter1 = LETTER;

         /* Prompt for user input and scan data. */
         printf("\nEnter one character, one integer, and another number> ");
         scanf("%c%d%lf", &letter2, &val1, &val2);
            …

         return (0);
      }
```

Section 2.6

2.   value = -3.6175   (# means blank)

```
      Format       Output
      %8.4f       #-3.6175
      %8.3f       ##-3.618
      %8.2f       ###-3.62
      %8.1f       ####-3.6
      %8.0f       #####-4.
      %.2f        -3.62
```

Programming

1.   `printf("%5d%11.2f%9.1f", a, b, c);`

Section 2.7

2.   In an interactive program the data is taken from keyboard input. In a batch program the input comes from a file.

Programming

1.

```
      /*
       * Gets three input characters which are user's initials and displays
       * them in a welcoming message.  Then gets input of the quantities of
       * each of the following coins, in the respective order, quarters,
       * dimes, nickels, and pennies.  Computes the total value of the
       * coins, and then displays the value.  Input is taken from a file
       * provided through input redirection.  Output can be redirected to
       * a file if desired.
       */

      #include <stdio.h>

      int
      main(void)
      {
         char first, middle, last; /* input - 3 initials              */
         int pennies, nickels;     /* input - count of each coin type */
         int dimes, quarters;      /* input - count of each coin type */
         int change;               /* output - change amount          */
         int dollars;              /* output - dollar amount          */
         int total_cents;          /* total cents                     */

         /* Get and display the customer's initials. */
         scanf("%c%c%c", &first, &middle, &last);
         printf("\nHello %c%c%c, let's see what your coins are worth.\n",
                 first, second, third);

         /* Get the count of each kind of coin. */
         scanf("%d", &quarters);
         printf("Number of quarters is %1d.\n", quarters);
         scanf("%d", &dimes);
         printf("Number of dimes is %1d.\n", dimes);
         scanf("%d", &nickels);
         printf("Number of nickels is %1d.\n", nickels);
```

```
   scanf("%d", &pennies);
   printf("Number of pennies is %1d.\n", pennies);

   /* Compute the total value in cents. */
   total_cents = 25 * quarters + 10 * dimes +
                 5 * nickels + pennies;

   /* Find the value in dollars and change. */
   dollars = total_cents / 100;
   change = total_cents % 100;

   /* Display the value in dollars and change. */
   printf("\nYour coins are worth %d dollars and %d cents.\n",
           dollars, change);*/

   return (0);
}

/*
 * Gets three input characters which are user's initials and displays
 * them in a welcoming message.  Then gets input of the quantities of
 * each of the following coins, in the respective order, quarters,
 * dimes, nickels, and pennies.  Computes the total value of the
 * coins, and then displays the value.  Input is taken from the file
 * defined as the macro MY_INPUT.  Output is directed to the file
 * defined as the macro MY_OUTPUT.
 */

#include <stdio.h>

/* These two lines define input and output file names to be used. */
#define MY_INPUT "prog2-7.in"
#define MY_OUTPUT "prog2-7.out"

int
main(void)
{
   char first, middle, last; /* input - 3 initials              */
   int pennies, nickels;     /* input - count of each coin type */
   int dimes, quarters;      /* input - count of each coin type */
   int change;               /* output - change amount          */
   int dollars;              /* output - dollar amount          */
   int total_cents;          /* total cents                     */
   FILE *inp;                /* input file pointer              */
   FILE *outp;               /* output file pointer             */

   /* Open the input and output files. */
   inp = fopen(MY_INPUT, "r");
   outp = fopen(MY_OUTPUT, "w");

   /* Get and display the customer's initials. */
   fscanf(inp, "%c%c%c", &first, &middle, &last);
   fprintf(outp, "\nHello %c%c%c, let's see what your coins are worth.\n",
           first, second, third);

   /* Get the count of each kind of coin. */
   fscanf(inp, "%d", &quarters);
   fprintf(outp, "Number of quarters is %1d.\n", quarters);
   fscanf(inp, "%d", &dimes);
   fprintf(outp, "Number of dimes is %1d.\n", dimes);
   fscanf(inp, "%d", &nickels);
   fprintf(outp, "Number of nickels is %1d.\n", nickels);
   fscanf(inp, "%d", &pennies);
   fprintf(outp, "Number of pennies is %1d.\n", pennies);

   /* Compute the total value in cents. */
   total_cents = 25 * quarters + 10 * dimes +
                 5 * nickels + pennies;

   /* Find the value in dollars and change. */
   dollars = total_cents / 100;
   change = total_cents % 100;

   /* Display the value in dollars and change. */
   fprintf(outp, "\nYour coins are worth %d dollars and %d cents.\n",
           dollars, change);*/
```

```
         fclose(inp);
         fclose(outp);

         return (0);
      }
```

Chapter 3 Top-Down Design with Functions

Section 3.1

```
2.    /* Computes employee gross salary. */

      #include <stdio.h>

      int
      main(void)
      {
         double hours;   /* input - number of hours worked */
         double rate;    /* input - hourly rate of pay     */
         double gross;   /* output - gross salary          */

         /* Get hours worked and rate of pay */

         /* Compute gross salary */
         /*    Assign hours x rate to gross */

         /* Display gross salary */

         return (0);
      }
```

Programming

```
1.    Algorithm Refinements
         Step 2 Refinement
         2.1 Assign  one + two   to   sum.
         Step 3 Refinement
         3.1 Assign  sum / 2   to   average.

      /*
       * Compute the sum and average of two numbers.
       */
      #include <stdio.h>

      int
      main(void)
      {
         double one, two,     /* input - numbers to process      */
            sum,              /* output - sum of one and two     */
            average;          /* output - average of one and two */

         /* Get two numbers. */
         printf("Enter two numbers separated by space> ");
         scanf("%lf%lf", &one, &two);

         /* Compute sum of numbers. */
         sum = one + two;

         /* Compute average of numbers. */
         average = sum / 2.0;

         /* Display sum and average. */
         printf("\nThe sum is %.2f, and the average is %.2f.\n", sum,
                average);

         return (0);
      }

2.    /*
       * Compute gross salary of employee given hours worked and hourly rate.
       */
```

```
#include <stdio.h>

int
main(void)
{
   double hours,        /* input - hours worked by employee */
      rate,             /* input - hourly rate of pay       */
      gross;            /* output - gross salary            */

   /* Get hours worked and rate of pay. */
   printf("Enter hours worked> ");
   scanf("%lf", &hours);
   printf("Enter hourly rate of pay> ");
   scanf("%lf", &rate);

   /* Compute gross salary. */
   gross = hours * rate;

   /* Display gross salary. */
   printf("\nThe gross salary is %.2f.\n", gross);

   return (0);
}
```

3.
```
/*
 * Compute gross salary of employee given regular hours worked, overtime
 * hours worked and hourly rate.
 */

#include <stdio.h>

int
main(void)
{
   double reg_hours,    /* input - regular hours worked  */
          ot_hours,     /* input - overtime hours worked */
          rate,         /* input - hourly rate of pay    */
          gross;        /* output - gross salary         */

   /* Get regular hours worked, overtime hours, and rate of pay. */
   printf("Enter regular hours worked> ");
   scanf("%lf", &reg_hours);
   printf("Enter overtime hours worked> ");
   scanf("%lf", &ot_hours);
   printf("Enter hourly rate of pay> ");
   scanf("%lf", &rate);

   /* Compute gross salary. */
   gross = reg_hours * rate + ot_hours * 1.5 * rate;


   /* Display gross salary. */
   printf("\nThe gross salary is %.2f.\n", gross);

   return (0);
}
```

4.
```
/*
 * Computes the weight of a batch of flat washers, the square
 * centimeters of material needed, and the weight of the "left-over"
 * material.
 * NOTE: The arrangement of washers on a sheet of material as used by
 *       this program is meant to be a pattern of side-by-side squares,
 *       where a square has a side dimension equal to the edge diameter
 *       of a washer.
 */

#include <stdio.h>
#include <math.h>

#define PI 3.14159

int
main(void)
{
   double hole_diameter; /* input - diameter of hole         */
```

```
         double edge_diameter; /* input - diameter of outer edge   */
         double thickness;     /* input - thickness of washer      */
         double density;       /* input - density of material used */
         double quantity;      /* input - number of washers made   */
         double weight;        /* output - weight of washer batch  */
         double hole_radius;   /* radius of hole                   */
         double edge_radius;   /* radius of outer edge             */
         double rim_area;      /* area of rim                      */
         double unit_weight;   /* weight of 1 washer               */
         double sheet_area;    /* sq cm of metal needed for batch  */
         double sheet_weight;  /* weight of sheet of needed area   */

         /* Get the inner diameter, outer diameter, and thickness. */
         printf("Inner diameter in centimeters> ");
         scanf("%lf", &hole_diameter);
         printf("Outer diameter in centimeters> ");
         scanf("%lf", &edge_diameter);
         printf("Thickness in centimeters> ");
         scanf("%lf", &thickness);

         /*
          * Get the material density and quantity manufactured.
          */
         printf("Material density in grams per cubic centimeter> ");
         scanf("%lf", &density);
         printf("Quantity in batch> ");
         scanf("%lf", &quantity);

         /* Compute the rim area. */
         hole_radius = hole_diameter / 2.0;
         edge_radius = edge_diameter / 2.0;
         rim_area = PI * edge_radius * edge_radius -
                    PI * hole_radius * hole_radius;

         /* Compute the weight of a flat washer. */
         unit_weight = rim_area * thickness * density;

         /* Compute the weight of the batch of washers. */
         weight = unit_weight * quantity;

         /* Compute the area of sheet of material needed. */
         sheet_area = edge_diameter * edge_diameter * quantity;

         /* Compute the weight of the sheet of material. */
         sheet_weight = sheet_area * thickness * density;

         /* Display the weight of the batch of washers. */
         printf("\nThe expected weight of the batch is %.2f ", weight);
         printf("grams.\n");

         /* Display the area of the sheet of material. */
         printf("\nThe sheet area needed is %.2f square cm.\n", sheet_area);

         /* Display the weight of the sheet of material. */
         printf("\nThe expected weight of the sheet is %.2f grams.\n",
                sheet_weight);

         /* Display the weight of the left-over material. */
         printf("\nThe expected weight of left-over material is %.2f ",
                sheet_weight - weight);
         printf("grams\n");

         return (0);
      }
```

<u>Section 3.2</u>

```
2.    a.    4.0
      b.    17.0
      c.    -8.0 * 9.0 = -72.0
      d.    22.0
      e.    2.828427
```

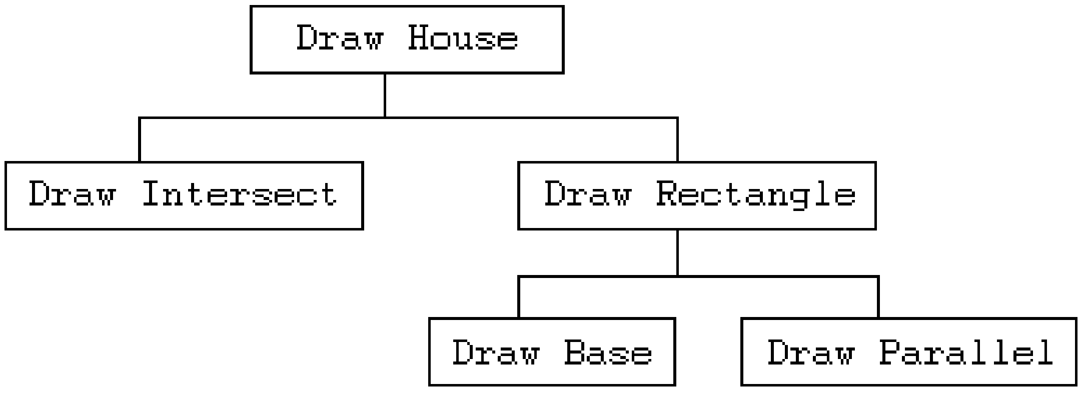
<u>Programming</u>

1.
```
#include <math.h>
double tmp, x, y;

tmp = fabs(x - y);
printf("%.2f", tmp) ;

          or

#include <math.h>
double tmp, x, y;

printf("%.2f", fabs(x - y));
```

2.
```
/*
 * Compute the distance between two 3-dimensional points.
 */

#include <stdio.h>
#include <math.h>

int
main(void)
{
   double x1, y1, z1, x2, y2, z2, distance;

   /* Get the coordinates. */
   printf("Enter the first set of x y z coordinates> ");
   scanf("%lf%lf%lf", &x1, &y1, &z1);
   printf("Enter the second set of x y z coordinates> ");
   scanf("%lf%lf", &x2, &y2, &z2);

   /* Compute the distance. */
   distance = sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2) + pow(z1 – z2, 2));

   /* Display the result. */
   printf("The distance between (%.2f,%.2f,%.2f) and (%.2f,%.2f, %.2f) is %.2f.\n",
           x1, y1, z1, x2, y2, z2, distance);

   return (0);
}
```

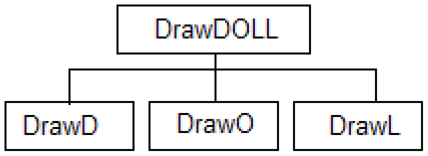
Section 3.3

2.



Section 3.4

2.



Programming

1.
```
/*
 * Draws parallel vertical lines.
 */
void
draw_parallel(void)
{
   printf("\n|      |") ;
   printf("\n|      |") ;
   printf("\n|      |") ;
}

/*
 * Draws a rectangle.
 */
void
draw_rectangle(void)
{
   draw_base();
   draw_parallel();
   draw_base();
}
```

2.
```
/*
 * Displays HI HO in block letters.
 */
#include <stdio.h>

/* function prototypes */
void print_i(void);
void print_o(void);
void print_h(void);

int
main(void)
{
   print_h();
   print_i();
   printf("\n\n\n");
   print_h();
   print_o();

   return (0);
}

/*
 * Displays I in block letter form.
 */
void
print_i(void)
{
   printf("******\n");
   printf("  **  \n");
   printf("  **  \n");
   printf("  **  \n");
   printf("******\n");
}

/*
 * Displays O in block letter form.
 */
void
print_o(void)
{
```

```
         printf("******\n");
         printf("**  **\n");
         printf("**  **\n");
         printf("**  **\n");
         printf("******\n");
      }

      /*
       * Displays H in block letter form.
       */
      void
      print_h(void)
      {
         printf("**  **\n");
         printf("**  **\n");
         printf("******\n");
         printf("**  **\n");
         printf("**  **\n");
      }
```

3.

```
      /*
       * Converts distances from miles to kilometers.
       */

      #include <stdio.h>                 /* printf, scanf definitions */
      #define KMS_PER_MILE 1.609    /* conversion constant       */

      /* function prototypes */
      void instruct(void);

      int
      main(void)
      {
         double miles,        /* distance in miles                 */
                kms;          /* equivalent distance in kilometers */

         /* Display instructions. */
         instruct();

         /* Get the distance in miles. */
         printf("Enter the distance in miles> ");
         scanf("%lf", &miles);

         /* Convert the distance to kilometers. */
         kms = KMS_PER_MILE * miles;

         /* Display the distance in kilometers. */
         printf("\nThat equals %f kilometers.\n", kms);

         return (0);
      }

      /*
       * Instruct users about the program.
       */
      void
      instruct(void)
      {
         printf("This program converts distances from miles to kilometers.");
         printf("\nTo use, enter distance in miles after prompt.\n");
      }
```

4.

```
      /*
       * Finds and displays the area and circumference of a circle.
       */

      #include <stdio.h>
      #define PI 3.14159

      /* function prototypes */
      void instruct(void);

      int
      main(void)
      {
         double radius,       /* input - radius of a circle         */
```

```
                area,         /* output - area of a circle          */
                circum;       /* output - circumference of a circle */

         /* Display instructions. */
         instruct();

         /* Get the circle radius */
         printf("Enter radius> ");
         scanf("%lf", &radius);

         /* Find the area */
         area = PI * radius * radius;

         /* Find the circumference */
         circum = 2 * PI * radius;

         /* Display the area and circumference */
         printf("The area is %.2f\n", area.;
         printf("The circumference is %.2f\n", circum);

         return (0);
      }

      /*
       * Instruct users about the program.
       */
      void
      instruct(void)
      {
         printf("This program computes the area\n");
         printf("and circumference of a circle.\n\n");
         printf("To use this program, enter the radius of\n");
         printf("the circle after the prompt: Enter radius>\n");
      }
```

Section 3.5

2. Since function scale multiplies its first argument(type double) by the power of 10 specified by its second argument (an integer), num_2 would be multiplied by 10 to the power of the whole number part of num_1.

Programming

1.
```
      /*
       * Computes the weight of a batch of flat washers.
       */

      #include <stdio.h>
      #include <math.h>

      #define PI 3.14159  /* Omit if PI defined in <math.h> */

      /* function prototypes */
      double find_area(double r);
      double find_rim_area(double outer, double inner);
      double find_unit_weight(double area, double thickness, double density);
      void instruct(void);

      int
      main(void)
      {
         double hole_diameter;      /* input - diameter of hole         */
         double edge_diameter;      /* input - diameter of outer edge   */
         double thickness;          /* input - thickness of washer      */
         double density;            /* input - density of material used */
         double quantity;           /* input - number of washers made   */
         double weight;             /* output - weight of washer batch  */
         double hole_radius;        /* radius of hole                   */
         double edge_radius;        /* radius of outer edge             */
         double rim_area;           /* area of rim                      */
         double unit_weight;        /* weight of 1 washer               */

         /* Give the user instructions. */
         instruct();

         /* Get the inner diameter, outer diameter, and thickness. */
```

```
   printf("Inner diameter in centimeters> ");
   scanf("%lf", &hole_diameter);
   printf("Outer diameter in centimeters> ");
   scanf("%lf", &edge_diameter);
   printf("Thickness in centimeters> ");
   scanf("%lf", &thickness);

   /* Get the material density and quantity manufactured. */
   printf("Material density in grams per cubic centimeter> ");
   scanf("%lf", &density);
   printf("Quantity in batch> ");
   scanf("%lf", &quantity);

   /* Compute the rim area. */
   hole_radius = hole_diameter / 2.0;
   edge_radius = edge_diameter / 2.0;
   rim_area = find_rim_area(edge_radius, hole_radius);

   /* Compute the weight of a flat washer. */
   unit_weight = find_unit_weight(rim_area, thickness, density);

   /* Compute the weight of the batch of washers. */
   weight = unit_weight * quantity;

   /* Display the weight of the batch of washers. */
   printf("\nThe expected weight of the batch is %.2f ", weight);
   printf("grams.\n");

   return (0);
}

/*
 * Displays instructions to a user of program to compute the
 * weight of a batch of flat washers.
 */
void
instruct(void)
{
   printf("This program computes the weight of a batch of flat \n");
   printf("washers.\n\n);
   printf("To use this program, please enter the inner diameter,\n");
   printf("outer diameter, thickness, density, and quantity at each\n");
   printf("respective prompt.\n\nThanks for using this program.\n\n");
}

/*
 * Computes the area of a circle with radius r.
 * Pre:  r is defined and is > 0.
 *       PI is a constant macro representing an approximation of pi.
 *       Library math.h is included.
 */
double
find_area(double r)
{
   return (PI * pow(r, 2));
}

/*
 * Computes the area of an annular ring with inner radius of inner
 * and outer radius of outer.
 * Pre:  inner and outer are defined and are > 0.
 *       Function find_area() is defined.
 */
double
find_rim_area(double outer, double inner)
{
   return (find_area(outer) - find_area(inner));
}

/*
* Computes the unit weight of a flat object with an area of area,
* with a thickness of thickness, and with a density of density.
* Pre:  area, thickness and density are defined and are > 0.
*/
double
find_unit_weight(double area, double thickness, double density)
```

```
      {
         return (area * thickness * density);
      }
```

2.
```
      /*
       * Computes the departure time required to reach a destination that
       * is a given (positive) distance away, based on supplied arrival
       * time and estimated average speed.  Arrival must be on same day
       * as departure.
       */
      #include <stdio.h>

      /* function prototype */
      double find_departure_time (int arr_time, double distance, double speed);

      int
      main(void)
      {
         int arr_time;        /* input--arrival time */
         double distance,     /* input--distance traveled (km) */
                avg_speed;    /* input--anticipated average speed (km/hr) */
         int dep_time;        /* output--required departure time */

         /* Get arrival time */
         printf("Enter arrival time as integer on a 24 hour clock.  For example,");
         printf("\n8:30 PM would be entered as 2030\n");
         printf("Arrival time> ");
         scanf("%d", &arr_time);

         /* Now get the distance to travel and anticipated average speed */
         printf("Enter the distance in km> ");
         scanf("%lf", &distance);

         printf("Enter anticipated average speed (including stops) in km/hr> ");
         scanf("%lf", &avg_speed);

         /* Compute and display the required departure time. */
         dep_time = find_departure_time(arr_time, distance, avg_speed);
         printf("You need to leave at %d.\n", dep_time );

         return (0);
      }

      /*
       * Returns the departure time required to travel distance
       *    given arrival time (arr_time) and speed.
       * Pre:  dist/speed <= arr_time converted to hours since midnight
       *    (i.e., arrival is same day as departure)
       */
      double
      find_departure_time (int arr_time, double distance, double speed)
      {
         double time;      /* travel time in hours */
         int tot_min,      /* travel time in minutes (rounded) */
             arr_min,      /* total minutes to arrival time */
             dep_tot_min,  /* departure time in minutes */
             dep_hr,       /* hour of departure (24-hr clock) */
             dep_min,      /* minutes of departure time */
             dep_time;     /* departure time (24-hr clock) */

         time = distance / speed;
         tot_min = int(time * 60 + 0.5);
         arr_min = arr_time / 100 * 60 + arr_time % 100;
         dep_tot_min = arr_min - tot_min;
         dep_hr = dep_tot_min / 60;
         dep_min = dep_tot_min % 60;
         dep_time = dep_hr * 100 + dep_min;
         return (dep_time);
      }
```

Chapter 4 Selection Structures: if and switch Statements

Section 4.2

```
2.    a = 6   b = 9    c = 14    flag = 1
                                                           Not evaluated
      a.    c == a + b || !flag           0 (FALSE)
      b.    a != 7 && flag || c >= 6      1 (TRUE)          c >= 6
      c.    !(b <= 12) && a % 2 == 0      0 (FALSE)         a % 2 == 0
      d.    !(a > 5 || c < a + b)         0 (FALSE)         c < a + b

4.    a.    c != a + b  && flag
      b.    (a == 7 || !flag) && c < 6
      c.    b <= 12) || a % 2 != 0
      d.    a > 5 || c < a + b
```

Programming

```
1.    a.    age >= 18 && age <= 21
      b.    water < 1.5 && water > 0.1
      c.    year % 4 == 0
      d.    speed <= 55
      e.    y > x && y < z
      f.    w == 6 || w <= 3

2.    a.    if (n < -k || n > k)            or    between = !(n < -k || n > k);
               between = 0;
            else
               between = 1;

      b.    if (digit == 0) /* check for divide by 0 before attempting division */
               divisor = 0;
            else if (num % divisor == 0)    or    else divisor = (num % divisor == 0);
               divisor = 1;
            else
               divisor = 0;

      c.    if (ch >= 'a' && ch <= 'z')     or    uppercase = ch >= 'a' && ch <= 'z');
               lowercase = 1;
            else
               lowercase = 0;
```

Section 4.3

```
2.    a.    x = 15.0
      b.    x = 50.0
      c.    x = 50.0
```

Programming

```
1.    a.    if (item != 0)
               product = item * product;
            printf("%.2f", product);

      b.    if ((x - y) < 0)
               y = y - x;
            else
               y = x - y;

            or

            y = x - y;
            if (y < 0) y = -y;

      c.    if (x == 0)
               zero_count = zero_count + 1;
            else
               if (x < 0)
                  minus_sum = minus_sum + x;
               else
                  plus_sum = plus_sum + x;
```

Section 4.4

```
2.    if (deduct < balance) {
         balance = balance – deduct;
         printf("New balance is %.2f\n", balance);
      } else {
         printf("Deduction of %.2f refused.\n", deduct);
         printf("Would overdraw account\n");
      }
      printf("Deduction = %.2f Final balance = %.2f",
         deduct, balance);
```

Programming

1.
```
      if (n > 0)
         printf("Average = %.2f\n", total / n);
      else
         printf("Error: The count of the numbers is zero.\n");
```

2.
```
      #include <stdio.h>

      /* Different type shapes to compute area for. */
      #define SELECT_CIRCLE 'C'     /* Type Circle */
      #define SELECT_SQUARE 'S'     /* Type Square */
      #define PI 3.14159265

      int
      main(void)
      {
            char type;        /* Type of shape. */
            double  area, side_base, radius;

            /* Display menu and get the type of shape to compute the area. */
            printf("Enter a:\n");
            printf("   %c -- To compute the area of a circle.\n",
                   SELECT_CIRCLE);
            printf("   %c -- To compute the area of a square.\n",
                   SELECT_SQUARE);
            printf("Select> ");
            scanf("%c", &type);

            /* Compute the area of different shapes. */
            if (type == SELECT_CIRCLE) {
                  /* Get information for CIRCLE radius needed to
                     compute the area. */
                  printf("Enter radius> ");
                  scanf("%lf", &radius);
                  area = PI * radius * radius;
                  printf("The area of the circle is %.4f\n", area);
            } else {
                  /* Get information for SQUARE edge needed to
                     compute the area. */
                  if (type == SELECT_SQUARE) {
                     printf("Enter side> ");
                     scanf("%d", &side_base);
                     area = side_base * side_base;
                     printf("The area of the square is %d\n", area);
                  } else {
                     printf("ERROR: Invalid selection.\n");
                  }
            }

            return (0);
      }
```

Section 4.5

2. Make these additions/changes to the Data Requirements section:

```
      Problem Inputs
      char shape              /* shape of washer                          */
      Replace hole_diameter and edge_diameter with:

      double inner_dim, /* width (dimension) of hole  */
      double outer_dim  /* outer diameter or width of one side */
```

Relevant Formulas
area of a square = side^2

Initial Algorithm changes:

```
1. Get the washer's shape, thickness, density, and quantity.
2. Get the washer's inner and outer dimensions.
(3. through 6. remain the same.)

Step 2 Refinements:
2.1  if shape is round
      get the washer's inner and outer diameter
2.2  else if shape is square
      get the washer's inner and outer widths.

Step 3 Refinement changes:
3.1  if shape is round
      3.1.1
            hole_radius is inner_dim / 2.0.
      3.1.2
            edge_radius is outer_dim / 2.0.
      3.1.3
            rim_area is PI x edge_radius x edge_radius -
            PI x hole_radius x hole_radius
3.2   else if shape is square
      3.2.1
            rim_area is outer_dim x outer_dim - inner_dim x inner_dim
```

Programming

1.
```c
#define UNDER_EXCESS 100
#define OVERUSE_CHG_RATE 2.0

/*
 * Computes use charge, including surcharge for excessive consumption
 * A doubled rate is applied for water use greater than UNDER_EXCESS.

 * Pre:  previous and current are defined.
 *       UNDER_EXCESS represents maximum water use which does not
 *          get a penalty, in thousands of gallons.
 */
double
comp_use_charge(int previous, int current)
{
   int    used;          /* thousands of gallons used this quarter */
   double use_charge;    /* charge for actual water use */

   used = current - previous;
   if (used > UNDER_EXCESS) {
      /* conservation requirements are not met */
      use_charge = (OVERUSE_CHG_RATE * used - UNDER_EXCESS) *
                    PER_1000_CHG;
      printf("Use charge is at %.2f times normal rate for usage\n",
              OVERUSE_CHG_RATE);
      printf("above %d units, which customer exceeds by %d units.\n",
              UNDER_EXCESS, used - UNDER_EXCESS);
   } else {
      use_charge = used * PER_1000_CHG; /* conservation requirements met */
   }
   return (use_charge);
}
```

Section 4.6

Programming

1.
```c
/*
 * Computes and prints a water bill given an unpaid balance and
 * previous and current meter readings.  Bill includes a demand
 * charge of $35.00, a use charge of $1.10 per thousand gallons
 * if conservation guidelines are met (use <= 95% of last year's).
 * The use charge is doubled if guidelines are not met, and a
 * surcharge of $2.00 is added if there is an unpaid balance.
 */

#include <stdio.h>

#define DEMAND_CHG        35.00 /* basic water demand charge        */
#define PER_1000_CHG      1.10  /* charge per thousand gallons used */
```

```
#define LATE_CHG          2.00  /* surcharge assessed on unpaid balance */
#define OVERUSE_CHG_RATE  2.0   /* double use charge as non-conservation penalty */
#define CONSERV_RATE      95    /* percent of last year's use allowed this year */

/* Function prototypes */
void instruct_water(void);
double comp_use_charge(int previous, int current, int use_last_year);
double comp_late_charge(double unpaid);
void display_bill(double late_charge, double bill, double unpaid);

int
main(void)
{
   int    previous;      /* input - meter reading from previous
                            quarter in thousands of gallons. */
   int    current;       /* input - meter reading from current quarter */
   int    use_last_year; /* input - use same quarter last year       */
   double unpaid;        /* input - unpaid balance of previous bill  */
   double bill;          /* output - water bill                      */
   double use_charge;    /* charge for actual water use              */
   double late_charge;   /* charge for non-payment of part of previous balance */

   /* Display user instructions. */
   instruct_water();

   /* Get data:  unpaid balance, previous and current meter readings,
      and last year's usage for same quarter */
   printf("Enter unpaid balance> $");
   scanf("%lf", &unpaid);
   printf("Enter previous meter reading> ");
   scanf("%d", &previous);
   printf("Enter current meter reading> ");
   scanf("%d", &current);
   printf("Enter last year's usage for same quarter as current> ");
   scanf("%d", &use_last_year);

   /* Compute use charge. */
   use_charge = comp_use_charge(previous, current, use_last_year);

   /* Determine applicable late charge.  */
   late_charge = comp_late_charge(unpaid);

   /* Figure bill. */
   bill = DEMAND_CHG + use_charge + unpaid + late_charge;

   /* Print bill. */
   display_bill(late_charge, bill, unpaid);

   return (0);
}

/*
 * Displays user instructions.
 */
void
instruct_water(void)
{
   printf("This program figures a water bill based on the demand\n");
   printf("charge ($%.2f) and a $%.2f per 1000 gallons use charge\n",
          DEMAND_CHG, PER_1000_CHG);
   printf("for water no more than %d% of the amount of water used\n",
          CONSERV_RATE);
   printf("in previous quarter and a $%.2f per 100 gallons use\n",
          OVERUSE_CHG_RATE);
   printf("charge.  A $%.2f surcharge is added to accounts with an\n",
          LATE_CHG);
   printf("unpaid balance.\n\n\n");
   printf("Enter unpaid balance, previous and current meter readings,\n");

   printf("and last year's usage for the same quarter as the current\n");
   printf("on separate lines after the prompts.\n");
   printf("Press <return> or <enter> after typing each number\n");
}

/*
 * Computes use charge, including surcharge for failure to conserve.
```

```
        * Pre:  previous, current, and use_last_year are defined.
        */
       double
       comp_use_charge(int previous, int current, int use_last_year)
       {
          int    used;          /* thousands of gallons used this quarter   */
          double use_charge;    /* charge for actual water use              */

          used = current - previous;
          if (used <= CONSERV_RATE / 100.0 * use_last_year)
          {
             use_charge = used * PER_1000_CHG;  /* conservation requirements met */
          } else {
             printf("Use charge is at %.2f times normal rate since use of \n",
             OVERUSE_CHG_RATE);
             printf("%d units exceeds %d percent of last year's\n",
             used, CONSERV_RATE);
             printf(" %d-unit use.\n", use_last_year);
             use_charge = used * OVERUSE_CHG_RATE * PER_1000_CHG;
          }

          return (use_charge);
       }

       /*
       * Computes late charge.
       * Pre:  unpaid is defined.
       */
       double
       comp_late_charge(double unpaid)
       {
          double late_charge;   /* charge for non-payment of part of previous balance */

          if (unpaid > 0.0)
             late_charge = LATE_CHG;  /* Assess late charge on unpaid balance. */
          else
             late_charge = 0.0;

          return (late_charge);
       }

       /*
       * Displays late charge late charge if any and bill.
       * Pre:  late_charge, bill, and unpaid are defined.
       */
       void
       display_bill(double late_charge, double bill, double unpaid)
       {
          if (late_charge > 0.0)
          {
             printf("\nBill includes $%.2f late charge on unpaid\n", late_charge,
             printf(" balance of $%.2f\n", unpaid);
          }
          printf("\nTotal due = $%.2f\n", bill);
       }
```

Section 4.7

2. The tax for all salaries less than $15000 including negative ones would be calculated with the formula 0.15 x salary. So if a salary were negative, it would produce a negative tax result.

Programming

1.
```
       if (noise_db > 110)
             printf("%d-decibel noise is uncomfortable.\n", noise_db);
       else if (noise_db > 90)
             printf("%d-decibel noise is very annoying.\n", noise_db);
       else if (noise_db > 70)
             printf("%d-decibel noise is annoying.\n", noise_db);
       else if (noise_db > 50)
             printf("%d-decibel noise is intrusive.\n", noise_db);
       else
             printf("%d-decibel noise is quiet.\n", noise_db);
```

```
2.    if (grade < 0.0)
            printf("Error\n");
      else if (grade < 1.99)
            if (grade <= .99)
                  printf("Failed semester -- registration suspended\n");
            else
                  printf("On probation for next semester\n");
      else if (grade >= 3.0)
            if (grade <= 3.49)
                  printf("Dean's list for semester\n");
      else if (grade <= 4.0)
            printf("Highest honors for semester\n");
      else
            printf("Error\n");

3.    if (wind_speed < 25)
            printf("Not a strong wind\n");
      else if (wind_speed <= 38)
            printf("Strong wind\n");
      else if (wind_speed <= 54)
            printf("Gale\n");
      else if (wind_speed <= 72)
            printf("Whole gale\n");
      else
            printf("Hurricane\n");

4.    /*
       *  This program categorizes systolic blood pressure readings
       */

      #include <stdio.h>
      #define HYPER_CUTOFF     140
      #define PRE_HYPER_CUTOFF 120

      int
      main(void)
      {
         int systolic;        /* systolic blood pressure in mmHg */

         /* Enter the systolic blood pressure reading. */
         printf("Enter the systolic blood pressure in mmHg> ");
         scanf("%d", &systolic);

         /* Categorize the reading. */
         if (systolic >= HYPER_CUTOFF)
            printf("hypertension\n");
         else if (systolic >= PRE_HYPER_CUTOFF)
            printf("pre-hypertension\n");
         else
            printf("normal\n");

         return (0);
      }
```

Section 4.8

2. In 4.16 there would be more than 10 values in each label set of an equivalent switch; in 4.17 the selector would be of type double.

Programming

```
1.    switch (watts) {
      case 15:
         lumens = 125;
         break;

      case 25:
         lumens = 215;
         break;

      case 40:
         lumens = 500;
         break;
      case 60:
         lumens = 880;
         break;

      case 75:
         lumens = 1000;
         break;

      case 100:
         lumens = 1675;
         break;

      default:
         printf("\nError: Unknown lumens.");
         lumens = -1;
      }
```

2.
```
      if (watts == 15)
         lumens = 125;
      else if (watts == 25)
         lumens = 215;
      else if (watts == 40)
         lumens = 500;
      else if (watts == 60)
         lumens = 880;
      else if (watts == 75)
         lumens = 1000;
      else if (watts == 100)
         lumens = 1675;
      else {
         printf("\nError: Unknown lumens.");
         lumens = -1;
      }
```

Chapter 5 Repetition and Loop Statements

Section 5.2

2.
```
0  2  4  6
```

Programming

1.
```
int i, value;

i = 0;
value = 1;
while (i < 7)
{
   printf("%3d %4d\n", i, value);
   i = i + 1;
   value = value * 2;
}
```

Section 5.3

2.
```
a. Enter an integer> 5
25
125
625
3125
b. Enter an integer> 6
36
216
1296
7776
c. Enter an integer> 7
49
343
2401
16807
```

4.
```
s /= 5;
```

```
      Not possible since (q * n) cannot be separated, and (n + 4) is not parenthesized.
      z -= x * y;
      t += u % v;
```

Programming

```
1.    /*
       * Computes the sum of integers from 1 to n inclusive.
       */
      #include <stdio.h>

      int
      main(void)
      {
         int n,         /* The last number to add, user input */
             sum,       /* Sum of the numbers.                */
             i;    /* Counter 1..n                       */

         printf("Enter the number n to compute the sum 1 + 2 + 3 + ... + n> ");
         scanf("%d", &n);

         i = 0;
         sum = 0;
         while (i < n) {
            i = i + 1;
            sum = sum + i;
         }

         printf("The sum of the integers 1 .. %d is %d.\n", n, sum);
         if ( sum == (n * (n + 1))/2 )
            printf("This equals ");  /* message 1 */
         else
            printf("This does not equal ");  /* message 2 */
         printf("(n * (n + 1)) / 2 for n = %d\n", n);

         return (0);
      }

      Message 1 will be displayed.
```

Section 5.4

```
 2.   a. Celsius    Fahrenheit
          -5          23.00
           0          32.00
           5          41.00
          10          50.00

      b. Celsius    Fahrenheit  (Loop repetition condition is false upon first test.)

      c. Celsius    Fahrenheit

      d. Celsius    Fahrenheit
          -5          23.00
           5          41.00

4.    n = 35; m = 12; p = 11;

6.    for (mult4 = 0;  mult4 <= 100;  mult4 += 4)
            printf("%d\n", mult4);
```

Programming

```
1.    #define PI 3.14159
               . . .

      for (i = init_degree;
           i <= final_degree;
           i = i + step_degree) {
         radians = i * PI / 180.0;
         printf("%d  %.5f  %.5f\n", i, sin(radians), cos(radians));
      }

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

```
      #define STEP 10
      #define CENT_TO_INCH 0.3937

      int
      main(void)
      {
         int cm,        /* Loop counter. */
             start,     /* Starting value of the table. */
             end;       /* Ending value of the table. */

         /* Get the starting and ending cm values. */
         printf("Enter the starting and ending centimeter values> ");
         scanf("%d%d", &start, &end);

         /* Display the conversion table. */
         printf("\n Centimeters            Inches\n");
         for (cm = start;  cm <= end;  cm += STEP) {
             printf("%7d        %15.0f\n", cm, cm * CENT_TO_INCH);
         }

         return (0);
      }
```

Section 5.5

2.
```
      for (sum = 0;  sum <= 100;  sum += 4)
         printf("%d\n", sum);
```

4.
```
      /*
       * Monitor gasoline supply in storage tank. Issue warning when supply
       * falls below MIN_PCT % of tank capacity.
       */

      #include <stdio.h>

      /* constant macros */
      #define CAPACITY  80000.0           /* number of barrels tank can hold */
      #define MIN_PCT        10           /* warn when supply falls below this percent of
                                             capacity */
      #define GALS_PER_BRL 42.0           /* number of U.S. gallons in one barrel */

      int
      main(void)
      {
        double min_supply,    /* minimum number of barrels left without warning */
              current,        /* current supply in barrels */
              remov_gals,     /* amount of current delivery in */
              remov_brls;     /*   barrels and gallons */
        int   count_deliv;    /* number of deliveries */

        /* Compute minimum supply without warning */
        min_supply = MIN_PCT / 100.0 * CAPACITY;

        /* Get initial supply and subtract amounts removed as long as minimum supply remains */
        printf("Number of barrels currently in tank> ");
        count_deliv = 0;
        for (scanf("%lf", &current); current >= min_supply; current -= remov_brls) {
            count_deliv++;
            printf("%.2f barrels are available.\n\n", current);
            printf("Enter number of gallons removed> ");
            scanf("%lf", &remov_gals);
            remov_brls = remov_gals / GALS_PER_BRL;
            printf("After removal of %.2f gallons (%.2f barrels),\n", remov_gals,
                remov_brls);
        }

        /* Issue warning. */
        printf("only %.2f barrels are left.\n\n", current);
        printf("The number of deliveries: %d\n\n", count_deliv);
        printf("*** WARNING ***\n");
        printf("Available supply is less than %d percent of tank's ", MIN_PCT);
        printf("%.2f-barrel capacity.\n", CAPACITY);

        return (0);
      }
```

The loop control variable is current.

Programming

1.
```
      /*
       * Computes and displays the annual population of a town; the initial
       * population is START_POP, the rate of annual increase is POP_INC,
       * and the population limit is MAX_POP.  The program will also
       * determine the number of years needed to exceed the population
       * limit and display that number of years along with the final
       * population.
       */
      #include <stdio.h>

      #define POP_INC   (10.0/100.0)      /* rate of population increase */
      #define START_POP 9870.0            /* starting population         */
      #define MAX_POP 30000.0             /* cutoff population           */

      int
      main(void)
      {
         int    count_year;
         double cur_pop;

         printf("Year  Population\n\n");
         count_year = 0;
         for (cur_pop = START_POP;
              cur_pop <= MAX_POP;
              cur_pop = cur_pop + cur_pop * POP_INC)
         {
            printf("%3d  %8.0f\n", count_year, cur_pop);
            count_year = count_year + 1;
         }

         printf("\nAfter %d years, population has exceeded %8.0f\n",
                   count_year, MAX_POP);
         printf("Final population = %8.0f\n", cur_pop);

         return (0);
      }
```

2.
```
      /*
       * Processes a payroll for a quantity of employees equal to
       * number_emp in a loop, and returns the payroll amount.
       * Pre:  number_emp is defined.
       */
      double
      do_payroll(int number_emp
      {
         double total_pay;          /* company payroll     */
         int    count_emp;          /* current employee    */
         double hours;              /* hours worked        */
         double rate;               /* hourly rate         */
         double pay;                /* pay for this period */

         /* Compute each employee's pay and add it to the payroll. */
         total_pay = 0.0:
         for (count_emp = 0;                    /* initialization            */
               count_emp < number_emp;    /* loop repetition condition */
               count_emp += 1)                  /* update                    */
         {
            printf("Hours> ");
            scanf("%lf", &hours);
            printf("Rate > $");
            scanf("%lf", &rate);
            pay = hours * rate;
            printf("Pay is $%6.2f\n\n", pay);
            total_pay = total_pay + pay;
         }
         /*
         printf("All employees processed\n");
         printf("Total payroll is $%8.2f\n", total_pay);
         */
         return (total_pay);
      }
```

Section 5.6

2. If the braces were omitted, only the printf statement would execute at each iteration; therefore the sum would not accumulate, no new input would be scanned, and the first score would be displayed infinitely.

Programming

1.
```
   /*
    * Computes n! using a translation of a given pseudocode.
    */
   #include <stdio.h>

   int
   main(void)
   {
      int n, p;

      scanf("%d", &n);
      p = 1;
      while (n > 0) {
         p *= n;
         --n;
      }

      printf("n! = %d\n", p);

      return (0);
   }
```

Label that gets displayed should be "n! = ".

2.
```
   /*
    * Compute the payroll for a company, stopping when SENT is entered
    * as an employee's hours.
    */

   #include <stdio.h>

   #define SENT -99

   int
   main(void)
   {
      double total_pay;    /* company payroll     */
      int    count_emp;    /* current employee    */
      int    number_emp;   /* number of employees */
      double hours;        /* hours worked        */
      double rate;         /* hourly rate         */
      double pay;          /* pay for this period */

      /* Compute each employee's pay and add it to the payroll. */
      total_pay = 0.0;
      count_emp = 0;
      printf("Hours> ");
      scanf("%lf", &hours);
      while (hours != SENT) {
         printf("Rate > $");
         scanf("%lf", &rate);
         pay = hours * rate;
         printf("\nPay is $%6.2f\n", pay);
         total_pay = total_pay + pay;
         count_emp = count_emp + 1;
         printf("Hours> ");
         scanf("%lf", &hours);
      }

      printf("All employees processed\n");
      printf("Total payroll is $%8.2f\n", total_pay);

      return (0);
      }
```

3.
```
      /*
       * Compute the payroll for a company, running in batch mode, with
       * input taken from a file provided through input redirection.
       * Output can be redirected to a file if desired.
       *
       * NOTE: Faulty data may produce some individual pay computations
       *       that are erroneous, since this program relies on end-of-file
       *       return from scanf() instead of trying to match pairs of
       *       hours and rate.
       */

      #include <stdio.h>

      int
      main(void)
      {
         double total_pay;    /* company payroll     */
         int    count_emp;    /* current employee    */
         int    input_status; /* Result of scanf     */
         double hours;        /* hours worked        */
         double rate;         /* hourly rate         */
         double pay;          /* pay for this period */


         /* Compute each employee's pay and add it to the payroll. */
         total_pay = 0.0:
         count_emp = 0;
         printf("Hours   Rate    Pay");
         input_status = scanf("%lf", &hours);
         while (input_status != EOF) {
            scanf("%lf", &rate);
            pay = hours * rate;
            printf("%6.2f   %6.2f   $%6.2f\n", hours, rate, pay);
            total_pay = total_pay + pay;
            count_emp = count_emp + 1;
            input_status = scanf("%lf", & hours);
         }

         printf("All employees processed\n");
         printf("Total payroll is $%8.2f\n", total_pay);

         return (0);
      }
```

3.
```
      int whole, frac;
      double num;

      whole = 0;
      frac = 0;
      printf("Enter a list of numbers terminated by zero");
      scanf("%lf", &num);
      while (num != 0) {
         if (int(num) == num)
            ++whole;
         else
            ++frac;
         scanf("%lf", &num);
      }
      printf("There were %d whole numbers and %d numbers with fractional parts.\n",
         whole, frac);
```

Section 5.7

2.
```
      Outer    0
         Inner   0  0
         Inner   0  1
         Inner   0  2
         Inner   0  1
      Outer    1
         Inner   1  0
         Inner   1  1
         Inner   1  2
         Inner   1  1
      Outer    2
         Inner   2  0
         Inner   2  1
         Inner   2  2
         Inner   2  1
```

Programming

1.
```
      /*
       * Displays a multiplication table from 0 to 9.
       */

      #include <stdio.h>
      #define NUM_DIGITS  10

      int
      main(void)
      {
         int  factor_1,    /* first factor         */
         factor_2,         /* second factor        */
         product;          /* product of factors   */

         /* Display the table heading.               */
         printf("\n*");
         for (factor_2 = 0;
              factor_2 < NUM_DIGITS;
              ++factor_2)
            printf("%3d", factor_2);  /* Display each digit */

         /* Display the table body.                  */
         for (factor_1 = 0;
              factor_1 < NUM_DIGITS;
              ++factor_1)
         {
            printf("\n%d", factor_1); /* Start a row with first factor. */
            for (factor_2 = 0;
                 factor_2 < NUM_DIGITS;
                 ++factor_2) {
               product = factor_1 * factor_2;
               printf("%3d", product);
            }
         }
         printf("\n");

         return (0);
      }
```

2.
```
      /*
       * Displays an arrangement of a series of numbers using nested loops.
       */
      #include <stdio.h>

      int
      main(void)
      {
         int i, j;

         /* display the upper half and the center row */
         for (i = 1; i <= 6; ++i)
         {
            for (j = 0; j < i; ++j)
            printf("%3d ", j);
            printf("\n");
         }

         /* display the lower half */
         for (i = 5; i > 0; --i)
         {
            for (j = 0; j < i; ++j)
               printf("%3d ", j);
               printf("\n");
         }

         return (0);
      }
```

Section 5.8

```
2.    sum = 0;
      odd = 1;
      do {
         sum += odd;
         odd += 2;
      } while (odd < n);
      printf("Sum of the positive odd numbers less than %d is %d\n", n, sum);
```

If n is less than 2, an incorrect result is displayed.

Programming

```
1.    int num1, num2;

      do {
         /* Get two numbers. */
         printf("Enter a pair of integers separated by space> ");
         scanf("%d%d", &num1, &num2);
      } while (num2 % num1 != 0);
```

Section 5.10

```
2.    sum = 0;
      k = 1;
      for (i = -n;  i < n - k;  ++i) {
         printf("DEBUG*** i=%d\n", i);
         sum += i * i;
         printf("DEBUG*** sum=%d\n", sum);
      }
```

Chapter 6 Modular Programming

Section 6.1

```
2.    sum_n_avg(one, two, three, &sum_of_3, &avg_of_3);
```

Programming

```
1.    /*
       * Computes sum and average of inputs n1, n2 and n3; returns sum
       * in parameter sump and returns average in parameter avgp.
       * Pre:  n1, n2, and n3 are defined.
       *       sump and avgp are addresses of memory cells (variables)
       *           capable of storing a double value.
       * Post: Sum of n1, n2, and n3 is stored in variable pointed to
       *           by sump.
       *       Average of n1, n2, and n3 is stored in variable pointed to
       *           by avgp.
       */
      void
      sum_n_avg(double  n1,     /* input numbers */
                double  n2,
                double  n3,
                double *sump,  /* output - sum of the three numbers */
                double *avgp)  /* output - average of the numbers   */
      {
         *sump = n1 + n2 + n3;
         *avgp = *sump / 3.0;
      }
```

Section 6.2

```
2.    x y z

   7   2   9
   7   2   9
  11   2   9
  18   2   9
  18   4   9
```

Programming

1.
```
/*
 * Simple program to illustrate use of a function sum().
 */
#include <stdio.h>

/* function prototypes */
int sum(int a, int b);

int
main(void)
{
   int x, y, z;

   x = 5; y = 3;

   printf("   x   y   z\n\n");
   z = sum(x, y);
   printf("%4d%4d%4d\n", x, y, z);

   z = sum(y, x);
   printf("%4d%4d%4d\n", x, y, z);

   x = sum(z, y);
   printf("%4d%4d%4d\n", x, y, z);

   x = sum(z, z);
   printf("%4d%4d%4d\n", x, y, z);

   y = sum(y, y);
   printf("%4d%4d%4d\n", x, y, z);

   return (0);
}

/*
 * Compute sum of two integers.
 * Pre: a and b are defined.
 */
int
sum(int a, int b)
{
   return (a + b);
}
```

Section 6.4

2. Output:
```
      x = 13, y = 7
```

The names of output parameters x and y do not end in p.

Section 6.5

2. Because get_operation should only return *, + , -, or /.

Programming

1.
```
#include <stdlib.h>

/*
 * Computes the GCD (Greatest Common Divisor) of inputs n1 and n2.
 * Pre:  n1 and n2 are defined.
 *       Library stdlib.h is included for abs() function.
 */
int
find_gcd(int n1, int n2)
{
   int q, p, r;

   q = abs(n1);
   p = abs(n2);
```

```
         r = q % p;         /* alternatively: a for loop with this header: */
         while (r != 0) {   /*  for  (r = q % p;  r != 0;  r = q % p)      */
            q = p;
            p = r;
            r = q % p;
         }
         return (p);
      }
```

2.

```
      /*
       * Multiplies fractions represented by pairs of integers.
       * Pre:  n1, d1, n2, d2 are defined;
       *       n_ansp and d_ansp are addresses of type int variables.
       * Post: product of n1/d1 and n2/d2 is stored in variables pointed
       *       to by n_ansp and d_ansp.  Result is not reduced.
       */
      void
      multiply_fractions(int   n1,     int   d1, /* input - first fraction  */
                         int   n2,     int   d2, /* input - second fraction */
                         int  *n_ansp,           /* output- product of two  */
                         int  *d_ansp)           /*         fractions       */
      {
         *n_ansp = n1 * n2;
         *d_ansp = d1 * d2;
         if (*d_ansp == 0)
         {
            printf("ERROR: Zero denominator.\n");
            *d_ansp = 1;
         }
      }
```

Chapter 7 Simple Data Types

Section 7.1

2. Underflow.

Programming

1. Results vary.

Section 7.2

2.

```
      int letter;
      for (letter = (int)'A';  letter <= (int)'A';  ++letter)
            printf("%c\n", (char)letter);
```

Programming

1.

```
      char
      next_char(char ch)
      {
         char result;
         result = (char)((int)ch + 1);
         return (result);
      }
```

2.

```
      char c;
      for (c = 'A'; c <= 'Z'; c = next_char(c))
            printf("%c\n", c);
```

Section 7.3

2.

	type definition	valid/invalid	flaw
a.	typedef enum {int, double, char} type_t;	invalid	Enumeration constants must be identifiers; they cannot be reserved words.
b.	typedef enum {p, q, r}	valid	

```
            letters_t;

            typedef enum                  invalid          Multiple declaration of the
               {o, p}                                      identifier p.
            more_letters_t;

      c.    typedef enum                  invalid          Enumeration constants must be
               {'X', 'Y', 'Z'}                             identifiers; they cannot be
            alpha_t;                                       characters.
```

Programming

```
1.    typedef enum
            {jan, feb, mar, apr, may, jun,
             jul, aug, sep, oct, nov, dec}
      month_t;

      month_t cur_month;

      if (cur_month == jan)
            printf("Happy New Year\n");
      else if (cur_month == jun)
            printf("Summer begins\n");
      else if (cur_month == sep)
            printf("Back to school\n");
      else if (cur_month == dec)
            printf("Happy Holidays\n");

      switch (cur_month) {
            case jan: printf("Happy New Year\n");
                      break;

            case jun: printf("Summer begins\n");
                      break;

            case sep: printf("Back to school\n");
                      break;

            case dec: printf("Happy Holidays\n");
                      break;
      }


2.    typedef enum
            {sun, mon, tue, wed, thu, fri, sat}
      day_t;

      int
      print_day(day_t cur_day)
      {
        switch (cur_day) {
            case sun: printf("Sunday\n");
                      break;

            case mon: printf("Monday\n");
                      break;

            case tue: printf("Tuesday\n");
                      break;

            case wed: printf("Wednesday\n");
                      break;

            case thu: printf("Thursday\n");
                      break;

            case fri: printf("Friday\n");
                      break;

            case sat: printf("Saturday\n");
        }
      }
```

Section 7.4

2. Take any function with a known whole-number root. For example,
   $f(x) = (x-4)(x+1) = x^2 - 3x - 4$
   has roots of 4 and -1. If bisect were called with this function and an interval whose midpoint was 4 ([2..6]) or -1 (-1.5..-0.5), then (f_mid == 0.0) would evaluate to 1.

Programming

1. Declare additional variable x_l:

```
double x_l;
```

After second scanf, if interval length is greater than 1, search for a 1-unit segment containing a root. If one is found, use it. Otherwise call bisect with entire interval.

```
if (x_right - x_left > 1) {
      for (x_l = x_left,
           x_l <= x_right - 1 && g(x_l) * g(x_l + 1) >= 0;
           x_l += 1) {}
      if (g(x_l) * g(x_l + 1) < 0) {
           x_left = x_l;
           x_right = x_l + 1;
      }
}
```

Chapter 8 Arrays

Section 8.1

2. There is room for 8 integers in array grades. The first integer element is referred to as list[0] and the last as list[7].

Section 8.2

```
2.                                                  Output (Note, output runs together since
      i = 2;                                              no \n in printfs)

      printf("%d  %.1f", 4, x[4]);                  4  12.0
      printf("%d  %.1f", i, x[i]);                  2  6.0
      printf("%.1f", x[i] + 1);                     7.0
      printf("%.1f", x[i] + i);                     8.0
      printf("%.1f", x[i + 1]);                     8.0
      printf("%.1f", x[i + i]);                     12.0
      printf("%.1f", x[2 * i]);                     12.0
      printf("%.1f", x[2 * i - 3]);                 12.0
      printf("%.1f", x[(int)x[4]]);                 Invalid.  Attempt to display x[12]
      printf("%.1f", x[i++]);                       6.0  (assigns 3 to i)
      printf("%.1f", x[--i]);                       6.0  (after assigning 2 to i)
      x[[i-1] = x[i];                               Assigns 6.0 (x[2]) to x[1]
      x[i] = x[i + 1];                              Assigns 8.0 to x[2]
      x[i] - 1 = x[1];                              Illegal assignment statement
```

Section 8.4

2.
```
int i, sum;

sum = 0;
for (i = 0; i < LIST_SIZE; ++i)
   if (list[i] % 2 == 1)
      sum += list[i];
```

Programming

1.
```
#include <stdio.h>
#define MAX_SIZE 10

int
main(void)
{
   int i, sum, x[MAX_SIZE];

   sum = 0;
   for (i = 0; i < MAX_SIZE; ++i) {
```

```
            scanf("%d", &x[i]);
            sum += x[i];
         }

         printf(" n     %% of total\n");
         for (i = 0; i < MAX_SIZE; ++i)
            printf("%2d%10.2f\n", (double)x[i] / sum);

         return (0);
      }
```

Section 8.5

```
2.    a.    Invalid: ar1 and ar2 are not declared as arrays.
      b.    Invalid: arrays c, d, and e have only 6 elements, so c[6] accesses an invalid memory
            location. Also, first three arguments must be arrays, not single values.
      c.    Valid: corresponding elements of c and d are added; sums are stored in e.
      d.    Invalid: arrays c, d, and e have only 6 elements. Function will use 7.
      e.    Valid: first 5 corresponding elements of c and d are added; sums are stored in first
            5 elements of e.
      f.    Invalid: passes a constant 6 as an array output argument.
      g.    Valid: corresponding elements of e and d are added; sums are stored in c.
      h.    Valid: all elements of c are doubled.
      i.    If c[1] is 4.3. Valid: 4 pairwise sums will be computed.  If c[1] is 91.7. Invalid:
            more than 6 elements will be used in the function.
      j.    Valid: because the first 3 arguments are pointers to the third element of their
            respective arrays, (subscript [2]), only the last 4 corresponding elements of c and d
            are added with results stored in last four elements of e ([2] through [5]).

4.    dbl_arr[i++] = data;
```

Programming

```
1.    /*
       * Computes product of first size elements of array a.
       */
      int
      multiply(const int a[], int size)
      {
         int product, i;

         product = 1;
         for (i = 0; i < size; ++i)
            product *= a[i];

         return (product);
      }

2.    #include <math.h>

      /*
       * Displays a table of the elements of array a and their absolute values.
       */
      void
      abs_table(const double a[], int size)
      {
         printf("   x      |x|\n");
         for (i = 0; i < size; ++i)
            printf("%8.1f  %8.1f\n", a[i], fabs(a[i]));
      }

3.    /*
       * Negates the first n elements of array a.
       */
      void
      negate(int a[], /* input/output - list of values to negate */
             int n)
      {
         int i;
         for (i = 0; i < n; ++i)
            a[i] = -a[i];
      }

4.    #include <stdlib.h>
```

```
      /*
       * Computes pairwise sums of first n corresponding
       * elements of arrays a and b.
       */
      void
      sum_correspond(int       c[], /* output - sums list */
                     const int a[], /* input */
                     const int b[], /* input */
                     int n)
      {
         int i;
         for (i = 0; i < n; ++i) {
            c[i] = a[i] + b[i];
         }
      }
```

5.

```
      void
      push(int  stack[], /* input/output - the stack */
           int  item,    /* input - data being pushed onto the stack */
           int *top,     /* input/output - pointer to top of stack */
           int  max_size)/* input - maximum size of stack */
      {
         if (*top < max_size - 1) {
             ++(*top);
             stack[*top] = item;
         }
      }

      int
      pop(int  stack[],  /* input/output - the stack */
          int *top);     /* input/output - pointer to top of stack */
      {
         int item;   /* value popped off the stack */

         if (*top >= 0) {
             item = stack[*top];
             --(*top);
         } else {
             item = STACK_EMPTY;
         }

         return (item);
      }

      int
      retrieve(const int stack[], /* input - the stack */
               int       top)     /* input - stack top subscript */
      {
         int item;

         if (top >= 0)
             item = stack[top];
         else
             item = STACK_EMPTY;

         return (item);
      }
```

Section 8.6

```
  2.   8  53  32  54  74   3
       3  53  32  54  74   8      (8 and 3 were exchanged)
       3   8  32  54  74  53      (53 and 8 were exchanged)
       3   8  32  54  74  53      (no exchanges)
       3   8  32  53  74  54      (54 and 53 were exchanged)
       3   8  32  53  54  74      (74 and 54 were exchanged)
      A total of 15 comparisons and 4 exchanges were made.

       7  18  29  37  42  42
      A total of 15 comparisons were made, but because the data are already
      in sorted order, no exchanges were necessary.
```

Programming

1.
```
/*
 *  Finds the position of the smallest element in the subarray
 *  list[first] through list[last] (inclusive).
 *  Pre:  first < last and elements 0 through last of array list are
 *           defined.
 *  Post: Returns the subscript small_sub of the smallest element in the
 *           subarray; i.e., list[small_sub] <= list[i] for all i in the
 *           subarray between first and last.
 */
int
get_min_range(int list[], int first, int last)
{
   int i,          /* Loop Control Variable (LCV)        */
   small_sub;      /* subscript of smallest value so far */

   small_sub = first;  /* Assume first element is smallest   */

   for (i = first + 1; i <= last; ++i)
      if (list[i] < list[small_sub])
         small_sub = i;

   return (small_sub);
}
```

2.
```
/*
 *  Finds the position of the largest element in the subarray
 *  list[first] through list[last] (inclusive).
 *  Pre:  first < last and elements 0 through last of array list are
 *           defined.
 *  Post: Returns the subscript large_sub of the largest element in the
 *           subarray; i.e., list[large_sub] >= list[i] for all i in the
 *           subarray between first and last.
 */
int
get_max_range(int list[], int first, int last)
{
   int i,          /* Loop Control Variable (LCV)       */
   large_sub;      /* subscript of largest value so far */

   large_sub = first;  /* Assume first element is largest. */

   for (i = first + 1; i <= last; ++i)
      if (list[i] > list[large_sub])
         large_sub = i;

   return (large_sub);
}

/*
 *  Sorts the data in array list.
 *  Pre:  first n elements of array list are defined.
 *        n >= 0.
 */
void
select_sort(int list[],  /* input/output - array being sorted   */
            int n)       /* input - number of elements to sort  */
{
   int temp;             /* temporary storage for an element of
                            list which is being swapped         */
   int fill;             /* last element in unsorted subarray   */
   int index_of_max;     /* subscript of next largest element   */

   for (fill = n - 1; fill > 0; --fill)
   {
      /* find position of largest element in unsorted subarray */
      index_of_max = get_max_range(list, 0, fill);

      /* exchange elements at fill and index_of_max */
      if (fill != index_of_max)
      {
         temp = list[index_of_max];
         list[index_of_max] = list[fill];
         list[fill] = temp;
      }
   }
}
```

3.
```
/*
 *  No changes at all for function get_min_range(), except to make
 *  list have the correct data type (double in this case) in the
 *  function header.
 */
int
get_min_range(double list[], int first, int last)
{
   int i,          /* Loop Control Variable (LCV)        */
   small_sub;      /* subscript of smallest value so far */

   /* executable statements can all be identical */
}

/*
 *  Function select_sort requires two changes - one is the data type
 *  of list in the function header (make it a double), and the other
 *  is the data type of the variable temp within the function body
 *  (should also be a double).
 */
void
select_sort(double list[],  /* input/output - array being sorted   */
            int n)          /* input - number of elements to sort  */
{
   double temp;             /* temporary storage for an element of
                               list which is being swapped         */
   int fill;                /* first element in unsorted subarray  */
   int index_of_min;        /* subscript of next smallest element  */

   /* executable statements can all be identical */
}
```

Section 8.7

Programming

1.
```
/*
 * Displays the values on the diagonal of matrix a
 */
void
print_diag(int a[10][10])
{
   int i;


   for (i = 0; i < 10; ++i)
      printf("Element(%d,%d): %d\n", i, i, a[i][i]);
}
```

Section 8.8

Programming

1.
```
/*
 * Sums each row separately, from column 0 to column (NUM_QUARTERS - 1).
 * Pre:  row_sum, sales, and num_rows are defined.
 *       row_sum has sufficient space to hold num_rows double values.
 *       sales has sufficient space to hold num_rows * NUM_QUARTERS
 *          double values.
 * Post: row_sum returns with the sum of each row in a separate array
 *          cell.
 */
void
sum_rows(double row_sum[],  /* output - returns sum of each row */
         double sales[][NUM_QUARTERS], /* input - matrix to be summed */
         int num_rows)      /* input - number of rows in sales */
{
   int i, j;               /* Loop Control Variables */

   for (i = 0; i < num_rows; ++i)
   {
      /* zero sum to start each row total */
      row_sum[i] = 0.0;
```

```
            /* sum across the row */
            for (j = 0; j < NUM_QUARTERS; ++j)
               row_sum[i] += sales[i][j];
         }
      }

      /*
       * Sums each column separately, from row 0 to row (num_rows - 1).
       * Pre:  col_sum, sales, and num_rows are defined.
       *       col_sum has sufficient space to hold NUM_QUARTERS double values.
       *       sales has sufficient space to hold num_rows * NUM_QUARTERS
       *          double values.
       * Post: col_sum returns with the sum of each column in a separate array
       *          cell.
       */
      void
      sum_columns(double col_sum[],  /* output - returns sum of each column */
                  double sales[][NUM_QUARTERS], /* input - matrix to be summed */
                  int num_rows)      /* input - number of rows in sales */
      {
         int i, j;               /* Loop Control Variables */

         for (i = 0; i < NUM_QUARTERS; ++i)
         {
            /* zero sum to start each column total */
            col_sum[i] = 0.0;

            /* sum down the column */
            for (j = 0; j < num_rows; ++j)
               col_sum[i] += sales[j][i];
         }
      }
```

2.

```
      typedef enum
         {no_winner, x_wins, 0_wins}
      status_code_t;

      /*
       * Checks for a winner in the tictac array, and returns a status code.
       * Pre:  tictac is defined.
       */
      status_code_t
      check_for_win(char tictac[][3]) /* input - array to be checked */
      {
         int i;               /* Loop Control Variable */

         /* first check all rows */
         for (i = 0; i < 3; ++i)
            if (tictac[i][0] == tictac[i][1] && tictac[i][0] == tictac[i][2])
               switch (tictac[i][0]) {
                  case 'x':
                  case 'X':
                     return (x_wins);
                     break;     /* never reach this line */

                  case 'o':
                  case 'O':
                     return (o_wins);
                     break;     /* never reach this line */

                  default :
                     /* means all spaces in row, so continue */
                     break;
               }

         /* next check all columns */
         for (i = 0; i < 3; ++i)
            if (tictac[0][i] == tictac[1][i] && tictac[0][i] == tictac[2][i])
               switch (tictac[0][i]) {
                  case 'x':
                  case 'X':
                     return (x_wins);
                     break;     /* never reach this line */

                  case 'o':
```

```
                  case 'O':
                     return (o_wins);
                     break;     /* never reach this line */

                  default :
                     /* means all spaces in column, so continue */
                     break;
               }

         /* next check main diagonal */
         if (tictac[0][0] == tictac[1][1] && tictac[0][0] == tictac[2][2])
            switch (tictac[0][0]) {
            case 'x':
            case 'X':
               return (x_wins);
               break;     /* never reach this line */

            case 'o':
            case 'O':
               return (o_wins);
               break;     /* never reach this line */

            default :
               /* means all spaces in diagonal, so continue */
               break;
            }

         /* lastly, check remaining diagonal */
         if (tictac[0][2] == tictac[1][1] && tictac[0][2] == tictac[0][2])
            switch (tictac[0][2]) {
            case 'x':
            case 'X':
               return (x_wins);
               break;     /* never reach this line */

            case 'o':
            case 'O':
               return (o_wins);
               break;     /* never reach this line */

            default :
              /* means all spaces in diagonal, so continue */
              break;
            }

         /* if function gets to this point, no winner was found */
         return (no_winner);
      }
```

Chapter 9 Strings

Section 9.1

2.   It displays the characters in the array until it reaches the null character.

Programming

1.

```
#include <stdio.h>

int
main(void)
{
   char in[25];

   for (scanf("%s", in);
        in[0] != '9';
        scanf("%s", in))
      printf("%s starts with the letter %c\n", in, in[0]);

   return (0);
}
```

Section 9.2

```
2.    a.  strncpy(ssnshort, socsec, 6);
          ssnshort[6] = '\0';
      b.  strncpy(ssn1, socsec, 3);
          ssn1[3] = '\0';
      c.  strncpy(ssn2, &socsec[4], 2);
          ssn2[2] = '\0';
      d.  strcpy(ssn3, &socsec[7]);
```

Programming

```
1.    /*
       *  Breaks down MMOC product code.
       */

      #include <stdio.h>
      #include <string.h>

      #define BUF_SZ   10  /* Scratch buffer size. */
      #define INPUT_SZ 30  /* Input buffer size.   */

      int
      main(void)
      {
         char buff[BUF_SZ], input[INPUT_SZ];
         int first, last;

         /*  Get data string representing product code. */
         printf("Enter a MMOC Product code > ");
         scanf("%s", input);

         /*  Find first digit */
         for (first = 0;
              ! (input[first] >= '0' && input[first] <= '9');
              ++first) {}

         strncpy(buff, input, first);
         buff[first] = '\0';
         printf("Warehouse:   %s\n", buff);

         /*  Find first capital letter following digits.  */
         for (last = first;
              input[last] < 'A' || input[last] > 'Z';
              ++last) {}

         strncpy(buff, &input[first], last - first);
         buff[last - first] = '\0';
         printf("Product:     %s\n", buff);

         printf("Qualifiers:  %s\n", &input[last]);

         return (0);
      }

2.    /*
       * Trims leading and trailing blanks from s.
       */
      char *
      trim_blanks (char       *trimmed,  /* output */
                   const char *to_trim)  /* input  */
      {
         int first, last;

         for (first = 0; to_trim[first] == ' '; ++first) {}
         for (last = strlen(to_trim) - 1; to_trim[last] == ' '; --last) {}
         strncpy(trimmed, &to_trim[first], last - first + 1);
         trimmed[last - first + 1] = '\0';
         return (trimmed);
      }
```

Section 9.3

2. The second argument to strcat (pres[7]) is a character rather than a string.
strncat(tmp1, &pres[7], 1); tmp1[8] = '\0';

```
      Value of tmp1:
            QUINCY J
```

Programming

```
1.    /*
       * Brackets a string according to its length.
       *         Length        Brackets used
       *         < 5 chars        <<   >>
       *         5 - 10 chars     (*   *)
       *         > 10 chars       /+   +/
       */
      char *
      bracket_by_len(char       *result, /* output */
                     const char *word,   /* input - string to bracket */
                     int         size)     /* input - maximum size of result */
      {
         int len;

         len = strlen(word);
         if (len + 5 > size) {
            printf("Insufficient space\n");
         } else if (len < 5) {
            strcpy(result, "<<");
            strcat(result, word);
            strcat(result, ">>");
         } else if (len < 11) {
            strcpy(result, "(*");
            strcat(result, word);
            strcat(result, "*)");
         } else {
            strcpy(result, "/+");
            strcat(result, word);
            strcat(result, "+/");
         }
         return (result);
      }
```

Section 9.4

Programming

```
1.    /*
       *  Finds the index of the string that comes first alphabetically in
       *  elements min_sub..max_sub of list
       *  Pre:  list[min_sub] through list[max_sub] are of uniform case;
       *        max_sub >= min_sub
       */
      int
      alpha_first(char list[][STR_SIZE], /* input - array of strings   */
                  int  min_sub, /* input - minimum and maximum subscripts */
                  int  max_sub) /*   of portion of list to consider       */
      {
            int first, i;

            first = min_sub;
            for  (i = min_sub + 1;  i <= max_sub;  ++i)
                if (strcmp(list[i], list[first]) < 0)
                      first = i;

            return (first);
      }

      /*
       *  Sorts the strings in array list in alphabetical order
       *  Pre:  first n elements of list reference strings of uniform case;
       *        n >= 0
       */
      void
      select_sort_str(char list[][STR_SIZ], /* input/output - array of strings
                                                 being ordered alphabetically */
                      int   n)      /* input - number of elements to sort      */
      {
            int   fill,      /* index of element to contain next string in order */
```

```
                  index_of_min;  /* index of next string in order */
            char  temp[STR_SIZ];

            for  (fill = 0;  fill < n - fill;  ++fill) {
                index_of_min = alpha_first(list, fill, n - 1);

                if (index_of_min != fill) {
                      strcpy(temp, list[index_of_min]);
                      strcpy(list[index_of_min], list[fill]);
                      strcpy(list[fill], temp);
                }
            }
      }
```

Section 9.5

2.   One cannot tell from the reference strs[4] whether strs is a two-dimensional array of characters or an array of pointers to strings.

Programming

1.
```
      /*
       *  Finds the index of the shortest string in elements min_sub..max_sub of list
       *  Pre:  max_sub >= min_sub
       */
      int
      shortest(char *list[], /* input - array of pointers to strings   */
               int        min_sub, /* input - minimum and maximum subscripts */
               int        max_sub) /*   of portion of list to consider       */
      {
            int short, i;

            short = min_sub;
            for  (i = min_sub + 1;  i <= max_sub;  ++i)
                if (strlen(list[i]) < strlen(list[short]))
                      short = i;

            return (short);
      }

      /*
       *  Orders the pointers in array list so they access strings
       *  from shortest to longest
       *  Pre:  first n elements of list reference strings;
       *        n >= 0
       */
      void
      select_sort_str(char *list[], /* input/output - array of pointers being
                                       ordered                                */
                      int   n)      /* input - number of elements to sort    */
      {
            int   fill,  /* index of element to contain next string in order */
                  index_of_short;  /* index of next string in order */
            char *temp;

            for  (fill = 0;  fill < n - fill;  ++fill) {
                index_of_short = shortest(list, fill, n - 1);

                if (index_of_short != fill) {
                      temp = list[index_of_short];
                      list[index_of_short] = list[fill];
                      list[fill] = temp;
                }
            }
      }
```

Section 9.6

Programming

1.
```
      /*
       * Scan a string whose size is less than dest_len.
       */
      char *
      scanstring(char *dest,    /* output - destination string  */
                 int   dest_len)  /* input  - destination size    */
      {
         int i, ch;

         if (dest_len > 0) {

            /* Skip over blanks. */
            while (isspace(c = getchar()) && c != EOF) {}

            /* Get the string. */
            if (ch == EOF) {
               dest[0] = '\0';
            } else {
               dest[0] = ch;
               i = 1;
               for (ch = getchar();
                    !isspace(ch)  &&  ch != EOF  &&  i < dest_len - 1;
                    ch = getchar())
                  dest[i++] = ch;
               dest[i] = '\0';
            }
         } else {
            printf("Warning: No space in destination string.\n");
         }

         return (dest);
      }
```

2.

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

      int
      main(void)
      {
         int ch, line, upper_cnt, punct_cnt, lower_cnt;

         line = 0;
         upper_cnt = 0;
         punct_cnt = 0;
         lower_cnt = 0;

         for (ch = getchar();
              ch != EOF;
              ch = getchar()) {
            putchar(ch);
            if (ch == '\n')
              ++line;
            else if (ispunct(ch))
              ++punct_cnt;
            else if (islower(ch))
              ++lower_cnt;
            else if (isupper(ch))
              ++upper_cnt;
         }
         printf("\nThe %d lines of text processed", line_cnt);
         printf(" contained %d capital letters,", upper_cnt);
         printf(" %d lowercase letters,\n", lower_cnt);
         printf("and %d punctuation marks.\n", punct_cnt);

         return (0);
      }
```

Section 9.7

2.

```
      char *digits[10]  = {"Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven",
                           "Eight", "Nine"};

      char buffer[80];
      double amount;

      ...
      sprintf(buffer, "%s and %d/100 dollars", digits[(int)amount],
              (int)(amount - (int)amount) * 100);
```

Programming

1.
```
      /*
       * Returns integer equivalent of string s.
       *   Pre:  s is a string representation of a valid integer
       */
      int
      strtoint(const char *s)
      {
         int result;
         sscanf(s, "%d", &result);
         return (result);
      }

      /*
       * Returns type double equivalent of string s.
       *   Pre:  s is a string representation of a valid type double value
       */
      double
      strtodouble(const char *s)
      {
         double result;
         sscanf(s, "%lf", &result);
         return (result);
      }
```

Chapter 10 Recursion

Section 10.1

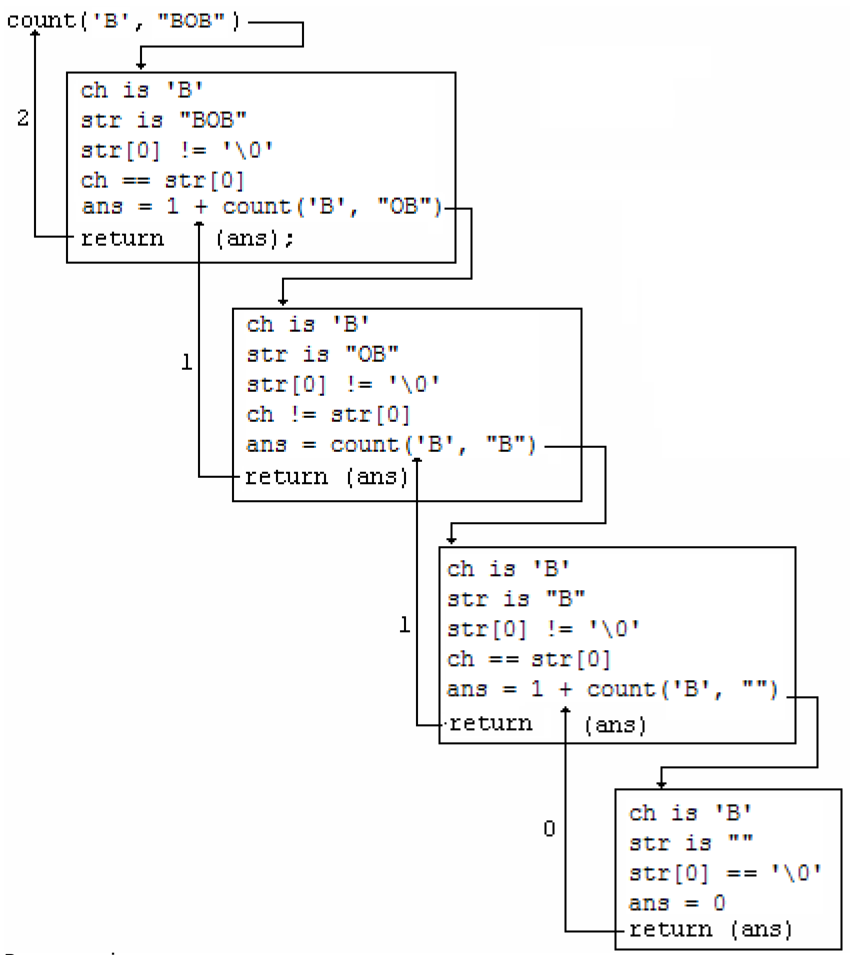
Programming

1.
```
      /*
       *  Counts the number of digits in the string str.
       */
      int
      count_digits(const char *str)
      {
            int ans;

            if (str[0] == '\0')              /*  simple case  */
                ans = 0;
            else                             /*  redefine problem using recursion */
                if (isdigit(str[0]))                 /*  first character must be counted */
                    ans = 1 + count_digits(&str[1]);
                else                                 /*  first character is not counted  */
                    ans = count_digits(&str[1]);

            return (ans);
      }
```

2.
```
      /*
       *  Performs integer addition.
       *  Pre:   m and n are defined and n > 0
       *  Post:  returns m + n
       */
      int
      add(int m, int n)
      {
            int ans;

            if (n == 0)
                  ans = m;     /* simple case */
            else
                ans = 1 + add(m, n-1);  /* recursive step */

            return (ans);
      }
```

Section 10.2

Activation frames of count('B',"BOB")



Programming

1.
```
/*
 *  Count the number of occurrences of the character ch in the string str.
 */
int
count(char ch, const char *str)
{
      int ans;

      printf("Entering count with ch = %c and str = %s\n", ch, str);
      if (str[0] == '\0')              /*  simple case  */
          ans = 0;
      else                             /*  redefine problem using recursion */
          if (ch == str[0])                  /*  first character must be counted   */
              ans = 1 + count(ch, &str[1]);

          else                               /*  first character is not counted   */
              ans = count(ch, &str[1]);

      printf("count(%c, %s) returning %d\n", ch, str, ans);

      return (ans);
}

Entering count with ch = l and str = lull
Entering count with ch = l and str = ull
Entering count with ch = l and str = ll
Entering count with ch = l and str = l
Entering count with ch = l and str =
count(l, ) returning 0
count(l, l) returning 1
count(l, ll) returning 2
count(l, ull) returning 2
count(l, lull) returning 3
```

Section 10.3

2. strange(7) = 2
strange(n) computes the integer portion of the log base 2 of n, i.e., $2^{strange(n)}$ = n if n is a power of 2. (log base 2 of 7 = 2.807)

Programming

1.
```
/*
 *  Computes (1 + 2 + 3 + ... (n-1) + n) using a recursive definition.
 *  Pre:  n > 0
 */
int
find_sum(int n)
{
      int ans;

      if (n == 1)
          ans = 1;
      else
          ans = n + find_sum(n - 1);

      return (ans);
}
```

2.
```
/*
 *  Computes and returns two Fibonacci numbers:
 *  fib(n+1) through fib_n1p, and fib(n) through fib_np
 */
void
fast_fib(int n, int *fib_n1p, int *fib_np)
{
   int fib_n, fib_n_1;

   if (n == 1) {
      *fib_n1p = 1;
      *fib_np = 1;
   } else {
      fast_fib(n - 1, &fib_n, &fib_n_1);
      *fib_n1p = fib_n + fib_n_1;
      *fib_np = fib_n;
   }
}
```

Section 10.4

2. Trace of recursive select sort:

```
Type in the number of items in the array - 4
Enter item 0 - 2
Enter item 1 - 12
Enter item 2 - 15
Enter item 3 - 1

Enter select_sort: n = 4 array = 2 15 12 1

   Enter place_largest: n = 4 array = 2 12 15 1
      max_index = 3
      max_index = 2
      max_index = 2
   max_index = 2
   Exit place_largest: n = 4 array = 2 12 1 15

Enter select_sort: n = 3 array = 2 12 1
```

```
         Enter place_largest: n = 3 array = 2 12 1
            max_index = 2
            max_index = 1
         max_index = 1
         Exit place_largest: n = 3 array = 2 1 12

      Enter select_sort: n = 2 array = 2 1

         Enter place_largest: n = 2 array = 2 1
            max_index = 1
         max_index = 0
         Exit place_largest: n = 2 array = 1 2

      Enter select_sort: n = 1 array = 1
      Exit select_sort: n = 1 array = 1
      Exit select_sort: n = 2 array = 1 2
      Exit select_sort: n = 3 array = 1 2 12
      Exit select_sort: n = 4 array = 1 2 12 15

      Array in ascending order :
      1 2 12 15
```

Programming

1.
```
      /*
       *  Forms a string containing all the digits found in the input
       *  parameter str.
       *  Pre:  digits has sufficient space to store all the digits in str plus the
       *        null string
       */
      char *
      find_digits(char  *digits,      /* output - string of all digits found in str    */
                  const char *str)    /* input  - string from which to extract digits  */
      {
            char restdigits[STRSIZ]; /* digits from reststr                       */

            if (str[0] == '\0')  {
                  digits[0] = '\0';  /* no characters in str => no digits in str     */
            } else {
                  if (isdigit(str[0]))
                        sprintf(digits, "%c%s", str[0],
                                find_digits(restdigits, &str[1]));
                  else
                        find_digits(digits, &str[1]);
            }

            return (digits);
      }
```

Section 10.5

Programming

1.
```
      /*
       *   Find the intersection of set1 and set2.
       */
      char *
      set_intersection(char     *result,   /* output - space in which to store string result */
                       const char *set1,   /* input  - sets whose intersection   */
                       const char *set2)   /*          is being formed           */
      {

            char temp[SETSIZ];       /* local variable to hold result of call to
                                        set_intersection embedded in sprintf call  */

            if (is_empty(set1))
                 result[0] = '\0';
            else if (is_element(set1[0], set2))
                 sprintf(result, "%c%s", set1[0],
                         set_intersection(temp, &set1[1], set2));
            else
                 set_intersection(result, &set1[1], set2);

            return (result);
      }
      /*
       *   Find the intersection of set1 and set2 (iterative version).
       */
      char *
      set_intersection(char *result,       /* output - space in which to store string result */
                       const char *set1,   /* input  - sets whose intersection     */
                       const char *set2)   /*         is being formed              */
      {
            int t,int_cnt;

            int_cnt = 0;

            for(t = 0;  t < strlen(set1); ++t)
                if (is_element(set1[t], set2))
                      result[int_cnt++] = set1[t];

            result[int_cnt] = '\0';

            return (result);
      }

      /*
       *  Determine if sets are equal.  If so, return 1;  if not, return 0.
       */
      int
      set_equal( const char  *set1,   /* input  - sets whose                     */
                 const char  *set2)   /*          equality is being determined  */
      {
            char set_temp[SETSIZ];

            return (strlen(set1) ==
                    strlen(set_intersection(set_temp, set1, set2)));
      }
```

Section 10.6

2.
```
      int
      main(void)
      {
            int  num_disks;

            printf("Type in number of disks: ");
            scanf("%d", &num_disks);
            printf("Moving %d disks from A to B using C\n", num_disks);
            tower('A', 'B', 'C', num_disks);

      return (0);
      }
```

Chapter 11 Structure and Union Types

Section 11.1

2.
```
      char*   "Hawaii"
      char       'W'
      resort.latitude.minutes
      char       'a'
```

Section 11.2

2.
```
      *locp                                              location_t
      (*locp).place or locp->place                       char*
      &(*locp).longitude or &locp->longitude             long_lat_t *
      (*locp).longitude.degrees or locp->longitude.degrees  int
```

Section 11.3

2. You could set the planet's name in the return value to the null string if get_planet fails or have the function also take an output argument that would be set to indicate success or failure.

Programming

1.
```
#include <stdio.h>

typedef struct {
     int numerator,
         denominator;
} frac_t;

/* Prototypes */
int gcd(int m, int n);
frac_t get_fraction(void);
void print_fraction(frac_t frac);
frac_t reduce_fraction(frac_t frac);

int
main(void)
{
     frac_t frac;

     printf("\n\nInput fraction (num/den): ");
     frac = get_fraction();
     print_fraction(frac);
     printf(" = ");
     print_fraction(reduce_fraction(frac));

     return (0);
}

/*
 *  Finds the greatest common divisor of m and n.
 *  Pre:  m and n are both > 0
 */
int
gcd(int m, int n)
{
      int ans;

      if (m % n == 0)
            ans = n;
      else
            ans = gcd(n, m % n);

      return (ans);
}

/*
 *  Scans and returns a fraction.
 */
frac_t
get_fraction(void)
{
     char slash;
     frac_t in_frac;

     scanf("%d%c%d", &in_frac.numerator, &slash, &in_frac.denominator);
     return (in_frac);
}

/*
 *  Displays a fraction with a slash between numerator and denominator.
 */
void
print_fraction(frac_t frac)
{
     printf("%d/%d", frac.numerator, frac.denominator);
}

/*
 * Returns a fraction reduced to lowest terms.
 */
frac_t
reduce_fraction(frac_t frac)
{
     frac_t red_frac;
```

```
          int nd_gcd;

          nd_gcd = gcd(frac.numerator, frac.denominator);
          red_frac.numerator = frac.numerator / nd_gcd;
          red_frac.denominator = frac.denominator / nd_gcd;

          return (red_frac);
     }
```

Section 11.4

Programming

```
1.    /*
       *  Returns product of complex values c1 and c2.
       */
      complex_t
      multiply_complex(complex_t c1, complex_t c2) /*  input parameters       */

      {
            complex_t cmult;

            cmult.real = c1.real * c2.real - c1.imag * c2.imag;
            cmult.imag = c1.real * c2.imag + c1.imag * c2.real;

            return (cmult);
      }

      /*
       *  Returns quotient of complex values (c1 / c2).
       */
      complex_t
      divide_complex(complex_t c1, complex_t c2) /*  input parameters         */
      {
            complex_t cdiv;
            double denom;

            denom =  c2.real * c2.real + c2.imag * c2.imag;
            cdiv.real = (c1.real * c2.real + c1.imag * c2.imag) / denom;
            cdiv.imag = (c1.imag * c2.real - c1.real * c2.imag) / denom ;


            return (cdiv);
      }
```

Section 11.5

```
2.    for (i = 0;  i < MAX_STU;  ++i)
         if (stulist[i].gpa + 0.2 < 4.0)
            stulist[i].gpa += 0.2;
         else
            stulist[i].gpa = 4.0;
```

Section 11.6

Programming

```
1.    /*
       *  Computes the perimeter of a figure given relevant dimensions.  Returns figure
       *  with perimeter component filled.
       */
      figure_t
      compute_perim(figure_t object)
      {
            switch (object.shape) {
            case 'C':
            case 'c':
                  object.fig.circle.circumference = 2 * PI * object.fig.circle.radius;
                  break;

            case 'R':
            case 'r':
                  object.fig.rectangle.perimeter = 2 * object.fig.rectangle.height
```

```
                                                 + 2 * object.fig.rectangle.width;
                  break;

            case 'S':
            case 's':
                  object.fig.square.perimeter = 4 * object.fig.square.side;
                  break;

            default:
                  printf("Error in shape code detected in compute_perim\n");
            }

            return (object);
      }

      /*
       *  Displays and labels the figure components.
       */
      void
      print_figure(figure_t object)
      {
            switch (object.shape) {
            case 'C':
            case 'c':
                  printf("Circle\n");
                  printf("Area          - %.3f\n", object.fig.circle.area);
                  printf("Circumference - %.3f\n", object.fig.circle.circumference);
                  printf("Radius        - %.3f\n\n", object.fig.circle.radius);
                  break;


            case 'R':
            case 'r':
                  printf("Rectangle\n");
                  printf("Area      - %.3f\n", object.fig.rectangle.area);
                  printf("Perimeter - %.3f\n", object.fig.rectangle.perimeter);
                  printf("Width     - %.3f\n", object.fig.rectangle.width);
                  printf("Height    - %.3f\n\n", object.fig.rectangle.height);
                  break;

            case 'S':
            case 's':
                  printf("Square\n");
                  printf("Area      - %.3f\n", object.fig.square.area);
                  printf("Perimeter - %.3f\n", object.fig.square.perimeter);
                  printf("Side      - %.3f\n\n", object.fig.square.side);
                  break;

            default:
                  printf("Error in shape code detected in print_figure\n");
            }
      }
```

Chapter 12 Text and Binary File Processing

Section 12.1

2.   fclose, fscanf, fprintf, getc, putc, fgets

Programming

1.
```
      #include <stdio.h>
      #define STRSIZ 80

      /* Prototypes */
      void copy(FILE *fpin, FILE *fpout);

      /*
       *  Makes a backup file.  Repeatedly prompts for the name of a file to back up until
       *  a name is provided that corresponds to an available file.  Then it prompts for
       *  the name of the backup file and creates the file copy.
       */
      int
      main(void)
      {
            char  in_name[STRSIZ],  /* strings giving names               */
                  out_name[STRSIZ]; /*    of input and backup files       */
            FILE *inp,              /* file pointers for input and        */
                 *outp;             /*    backup files                    */

            /*  Get the name of the file to back up and open the file for input.      */
            printf("Enter name of file you want to back up> ");
            for  (scanf("%s", in_name);
                  (inp = fopen(in_name, "r")) == NULL;
                  scanf("%s", in_name)) {
                printf("Cannot open %s for input\n", in_name);
                printf("Re-enter file name> ");
            }

            /*  Get name to use for backup file and open file for output.             */
            printf("Enter name for backup copy> ");
            scanf("%s", out_name);
            outp = fopen(out_name, "w");

            /*  Make backup copy                    */
            copy(inp, outp);

            /*  Close files and notify user of backup completion.                     */
            fclose(inp);
            fclose(outp);
            printf("Copied %s to %s.\n", in_name, out_name);

            return (0);
      }

      /*
       *  Makes a backup of TEXT file pointed to by fpin to fpout one char at a time.
       */
      void
      copy(FILE *fpin, FILE *fpout)
      {
            int ch;

            for  (ch = getc(fpin);  ch != EOF;  ch = getc(fpin))
                  putc(ch, fpout);
      }
```

Section 12.2

2. `fscanf(psn_txt_inp, "%s", exec.name);`

4. `fwrite(num_err, sizeof (int), SIZE, nums_outp);`

6. `fprintf(psn_txt_outp, "%s %d %f\n", exec.name, exec.age, exec.income);`

Programming

1.
```
      #include <stdio.h>
      #include <string.h>

      #define NAME_LEN    30       /* storage allocated for a unit name              */
      #define ABBREV_LEN  15       /* storage allocated for a unit abbreviation      */
      #define CLASS_LEN   20       /* storage allocated for a measurement class      */

      typedef struct {      /* unit of measurement type                        */
            char   name[NAME_LEN];    /* character string such as "milligrams"        */
            char   abbrev[ABBREV_LEN];/* shorter character string such as "mg"        */
            char   class[CLASS_LEN];  /* character string "volume","distance",or "weight"*/
            double standard;          /* number of standard units equivalent to this unit*/
      } unit_t;

      /*
       *  Gets data to place in units until value of sentinel is encountered in the name
       *  component of the input structure. Stops input prematurely if there are more than
       *  unit_max data values before the sentinel or if invalid data is encountered.
       *  Pre:  sentinel and unit_max are defined and unit_max is the declared size of
       *  units
       */
      void
      fread_units(int         unit_max,   /* input - declared size of unit    */
                  unit_t      units[],    /* output - array of data           */
                  int        *unit_sizep) /* output - number of data values
                                             stored in units          */
      {
            FILE   *fpdata;
            char   dataname[30];
            unit_t data;
            int    i, status;

            /* Sentinel input loop                                     */
            printf("Type in the name of the units database: ");
            scanf("%s", dataname);
            fpdata = fopen(dataname, "r");
            if (fpdata==NULL) {
                *unit_sizep = 0;
                printf("\nError opening %s !\n", dataname);
            } else {
                *unit_sizep = fread(units, sizeof(unit_t), unit_max, fpdata);
                fclose(fpdata);
            }
      }
```

Section 12.3

2. match determines if a record matches
   show displays a matching record

Programming

1.
```
      /*
       *  Displays each field of prod.  Leaves a blank line after the product
       *  display.
       */
      void
      show(product_t prod)
      {
            printf("Stock number - %d\n", prod.stock_num);
            printf("Category     - %s\n", prod.category);
            printf("Tech Descr   - %s\n", prod.tech_descript);
            printf("Price        - %.2lf\n\n", prod.price);
      }

      /*
       *  Prompts the user to enter the search parameters.
       */
      search_params_t
      get_params(void)
      {
            search_params_t sp = {MIN_STOCK, MAX_STOCK, "aaaa", "zzzz", "aaaa", "zzzz",
                                  0, MAX_PRICE};
            for  (response = menu_choose(sp);
                  response != 'q';
                  response = menu_choose(sp)) {
               switch(response) {

               case 'a':
                     printf("New low bound for stock number> ");
                     scanf("%d", &sp.low_stock);
                     break;

               case 'b':
                     printf("New high bound for stock number> ");
                     scanf("%d", &sp.high_stock);
                     break;

               case 'c':
                     printf("New low bound for category> ");
                     scanf("%s", sp.low_category);
                     break;

               case 'd':
                     printf("New high bound for category> ");
                     scanf("%s", sp.high_category);
                     break;

               case 'e':
                     printf("New low bound for technical description> ");
                     scanf("%s", sp.low_tech_descript);
                     break;

               case 'f':
                     printf("New high bound for technical description> ");
                     scanf("%s", sp.high_tech_descript);
                     break;

               case 'g':
                     printf("New low bound for price> ");
                     scanf("%lf", &sp.low_price);
                     break;

               case 'h':
                     printf("New high bound for price> ");
                     scanf("%lf", &sp.high_price);
                     break;

               default:
                     printf("The letter %c is not valid.\n", response);
                     printf("Please type a letter between a and h or q.\n");
               }
            }

            return (sp);
      }
```

2.
```
      /*
       * Converts a text file of product data to a binary file of product structures.
       */
      void
      make_product_file(FILE *fpin,    /*  file pointer to text input file */
                        FILE *fpout)   /*  file pointer to binary ouput file */
      {
        product_t p;
        int status;
        for(status = fscanf(fpin, "%d%s%s%lf",  &p.stock_num,
                            p.category, p.tech_descript, &p.price);
            status == 4;
            status = fscanf(fpin, "%d%s%s%lf", &p.stock_num,
                            p.category, p.tech_descript, &p.price)) {

           fwrite(&p, sizeof(p), 1, fpout);
         }
      }
```

Chapter 13 Programming in the Large

Section 13.2

2. prototype, block comment

Programming

1.
```
      /*
       * Myops.h - header file
       * Library of mathematical functions
       *
       * fabs - floating point absolute value
       * sqrt - square root
       * pow -  raises base to its exponent
       * factorial - finds factorial
       */

      /*
       * Returns the absolute value of the type double input argument x.
       */
      extern double
      fabs(double x);     /* input argument */

      /*
       * Returns the square root of the type double input argument x.
       *   Pre:  x >= 0.0
```

```
 */
extern double
sqrt(double x);     /* input argument where x must be >= 0.0 */

/*          y
 * Returns x  .
 *   Pre:  If x < 0 , y must be a whole number
 */
extern double
pow(double x,        /* input - base */
    double y);       /* input - exponent */

/*
 * Returns x!
 *   Pre:  x >= 0
 */
extern int
factorial(int x);    /* input argument where x >= 0 */
```

(The implementation of factorial could be either iterative or recursive.)

Section 13.3

2. One assumes that red is a system library and that blue is a personal library.

Programming

1.
```
/*
 * "complex.h" - header file
 * Library of complex number operators
 *
 * Type complex_t has these components -
 *   real, imag
 *
 * These are the operators -
 *  scan_complex - scans complex number from standard input
 *  print_complex - prints complex number to standard output
 *  add_complex - adds two complex numbers
 *  subtract_complex - subracts two complex numbers
 *  multiply_complex - multiplies to complex numbers
 *  divide_complex - divides two complex numbers
 *  abs_complex - finds magnitude of a complex number
 */

typedef struct complex_s {
      double real, imag;
} complex_t;

/*
 *  Complex number input function returns standard scanning error code:
 *    1 => valid scan,  0 => error,  negative EOF value => end of file
 */
extern int
scan_complex(complex_t *c); /*  output  -  address of complex variable to fill  */

/*
 *  Complex output function prints value as  (a + bi)  or  (a - bi),
 *  dropping a or b if they round to 0 unless both round to 0
 */
extern void
print_complex(complex_t c); /*  input  -  complex number to print        */

/*
 *  Returns sum of complex values c1 and c2.
 */
extern complex_t
add_complex(complex_t c1, complex_t c2); /*  input - values to add       */

/*
 *  Returns difference c1 - c2.
 */
extern complex_t
subtract_complex(complex_t c1, complex_t c2); /*  input parameters       */

/*
```

```
 *  Returns product of complex values c1 and c2.
 */
extern complex_t
multiply_complex(complex_t c1, complex_t c2); /*  input parameters       */

/*
 *  Returns quotient of complex values (c1 / c2).
 */
extern complex_t
divide_complex(complex_t c1, complex_t c2); /*  input parameters         */

/*
 *  Returns absolute value of complex number c.
 */
complex_t
abs_complex(complex_t c); /*  input parameter                            */

/*
 *  Implementation file "complex.c"
 *
 *  Operators to process complex numbers
 */
#include <stdio.h>
#include <math.h>
#include "complex.h"

/*
 *  Complex number input function returns standard scanning error code:
 *    1 => valid scan,  0 => error,  negative EOF value => end of file
 */
int
scan_complex(complex_t *c) /*  output  -  address of complex variable to fill  */
{
      int status;

      status = scanf("%lf%lf", &(*c).real, &(*c).imag);

      if (status == 2)
            status = 1;

      return (status);
}

/*
 *  Complex output function prints value as  (a + bi)  or  (a - bi),
 *  dropping a or b if they round to 0 unless both round to 0.
 */
void
print_complex(complex_t c) /*  input  -  complex number to print        */
{
      double a, b;
      char   sign;

      a = c.real;
      b = c.imag;

      printf("(");

      if (fabs(a) < .005  &&  fabs(b) < .005) {
            printf("%.2f", 0.0);
      } else if (fabs(b) < .005) {
            printf("%.2f", a);
      } else if (fabs(a) < .005) {
            printf("%.2fi", b);
      } else {
            if (b < 0)
                  sign = '-';
            else
                  sign = ' + ';
            printf("%.2f %c %.2fi", a, sign, fabs(b));
      }

      printf(")");
}

/*
```

```
       *  Returns sum of complex values c1 and c2.
       */
      complex_t
      add_complex(complex_t c1, complex_t c2) /*  input - values to add       */
      {
            complex_t csum;

            csum.real = c1.real + c2.real;
            csum.imag = c1.imag + c2.imag;

            return (csum);
      }

      /*
       *  Returns difference c1 - c2.
       */
      complex_t
      subtract_complex(complex_t c1, complex_t c2) /*  input parameters       */
      {
            complex_t cdiff;

            cdiff.real = c1.real - c2.real;
            cdiff.imag = c1.imag - c2.imag;


            return (cdiff);
      }

      /*
       *  Returns product of complex values c1 and c2.
       */
      complex_t
      multiply_complex(complex_t c1, complex_t c2) /*  input parameters       */
      {
            complex_t cmult;

            cmult.real = c1.real * c2.real - c1.imag * c2.imag;
            cmult.imag = c1.real * c2.imag + c1.imag * c2.real;

            return (cmult);
      }

      /*
       *  Returns quotient of complex values (c1 / c2).
       */
      complex_t
      divide_complex(complex_t c1, complex_t c2) /*  input parameters         */
      {
            complex_t cdiv;
            double denom;

            denom =  c2.real * c2.real + c2.imag * c2.imag;
            cdiv.real = (c1.real * c2.real + c1.imag * c2.imag) / denom;
            cdiv.imag = (c1.imag * c2.real - c1.real * c2.imag) / denom ;

            return (cdiv);
      }

      /*
       *  Returns absolute value of complex number c.
       */
      complex_t
      abs_complex(complex_t c) /*  input parameter                            */
      {
            complex_t cabs;

            cabs.real = sqrt(c.real * c.real + c.imag * c.imag);
            cabs.imag = 0;

            return (cabs);
      }
```

Section 13.4

2.   i

Section 13.6

2. a. The preprocessor will include the body of the #if directive, so the name PLANET_H_INCL as well as the types and operators will be defined by the code passed to the compiler.
   b. Since PLANET_H_INCL has been defined, the preprocessor will not include the body of the #if directive, so there will be no attempt to redefine the types and operators.

Section 13.7

Programming

1.
```
/*
 * This program sums the data in the input file named on the command line.
 * If invalid data is encountered, the program is terminated and the invalid
 * data is displayed.
 */

#include <stdio.h>

int
main(int   argc,
     char *argv[])
{
  FILE *finp;
  char charnum[20];
  int num, status, sum = 0;

  if (argc < 2) {
     printf("\nPlease include an input filename.\n");
     exit(1);
  }
  finp = fopen(argv[1], "r");
  if (finp == NULL) {
     printf("\nUnable to open input file.\n");
     exit(1);
  }

  for  (status = fscanf(finp, "%s", charnum);
        status == 1;
        status = fscanf(finp, "%s", charnum)) {
     if (sscanf(charnum, "%d", &num) != 1){
        printf("\nInvalid data found in %s - %s\n", argv[1], charnum);
        exit(1);
     } else {
        sum += num;
     }
  }
  printf("\nSum of %s is %d.\n", argv[1],  sum);

  return (0);
}
```

Section 13.8

2.
```
y = y - DOUBLE(p) -> y = y - (p) + (p)
NOT OK, the macro should have been written -
#define DOUBLE(x) ((x) + (x))
```

4.
```
PRINT_PRODUCT(a + b, a - b); ->
printf("%.2f X %.2f = %.2f\n", (a + b), (a - b), (a + b) * (a - b));
OK
```

Programming

1.
```
#define F_OF_X(x) (pow((x), 5) - 3 * pow((x), 3) + 4)
```

2.
```
#define PRINT_DOLLAR(x) printf("$%.2f", (x))
```

Chapter 14 Dynamic Data Structures

Section 14.2

2.
```
scanf("%d", nump);
```

4.
```
nump = (int *)calloc(12, sizeof (int));
```

Section 14.3

```
2.    (scale_node_t *)malloc(sizeof (scale_node_t));
      newp->
      newp
      prevp->linkp
```

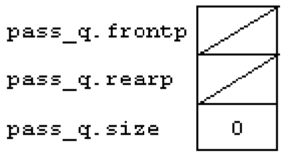
Section 14.4

Programming

```
1.    /* Return the length of the list pointed to by l. */
      int
      len_list(list_node_t *l)
      {
            if (l == NULL)
                  return(0);
            else
                  return(1 + len_list(l->linkp));
      }

2.    list_node_t *
      search(list_node_t *headp, /* input - pointer to head of list */
             int         target) /* input - value to search for */
      {
            if (headp == NULL) {
                  return(NULL);
            } else if (headp->digit == target) {
                  return(headp);
            } else {
                  return(search(headp->restp, target));
            }
      }
```

Section 14.6

2.



Section 14.7

```
2.    /*
       * Insert a new node containing new_key in order in old_list if the key is not already
       * present in the list.  Return the success or failure of the insertion by passing a 1
       * or a 0 back through the output parameter resultp.
       */
      list_node_t *
      insert_in_order(list_node_t *old_listp,   /* input/output */
                      int            new_key,   /* input        */
                      int           *resultp)   /* output       */
      {
            list_node_t *new_listp;

            if (old_listp == NULL) {
                  new_listp = (list_node_t *)malloc(sizeof (list_node_t));
                  new_listp->key = new_key;
                  new_listp->restp = NULL;
                  (*resultp) = 1;   /* success */
            } else if (old_listp->key > new_key) {
                  new_listp = (list_node_t *)malloc(sizeof (list_node_t));
                  new_listp->key = new_key;
                  new_listp->restp = old_listp;
                  (*resultp) = 1;   /* success */
            } else if (old_listp->key == new_key) {
                  new_listp = old_listp;
                  (*resultp) = 0;   /* failure */
            } else {
                  new_listp = old_listp;
                  new_listp->restp = insert_in_order(old_listp->restp, new_key, resultp);
            }

            return (new_listp);
      }
```

Programming

1.
```
      /*
       * Display the elements in the list pointed to by the pointer list.
       */
      void
      print_list(ordered_list_t *listp)
      {
            list_node_t *tmp;

            for(tmp = listp->headp;
                tmp != NULL;
                tmp = tmp->restp)
                  printf("%d\n", tmp->key);
      }
```

2.
```
      /*
       * Search for the value of key in the list.  If the value is not found return
       * NULL.
       */
      list_node_t
      retrieve_node(ordered_list_t *listp, int target)
      {
            list_node_t *tmp;

            for (tmp = listp->headp;
                 (tmp != NULL) && (tmp->key != target);
                 tmp = tmp->restp);

            return(tmp);
      }
```

Section 14.8

2.
```
      Display right subtree of node 40.
        Display left subtree of node 50.
          Display left subtree of node 45.
            Tree is empty -- return from displaying left subtree of node 45.
          Display item with key 45.
          Display right subtree of node 45.
            Tree is empty -- return from displaying right subtree of node 45.
          Return from displaying left subtree of node 50.
        Display item with key 50.
        Display right subtree of node 50.
          Display left subtree of node 65.
            Tree is empty -- return from displaying left subtree of node 65.
          Display item with key 65.
          Display right subtree of node 65.
            Tree is empty -- return from displaying right subtree of node 65.
          Return from displaying right subtree of node 50.
      Return from displaying right subtree of node 40.
```

4. 10, 12, 15, 25, 45, 55, 60

Programming

1.
```
      /*
       * Display the keys of the binary tree inorder.
       */
      void
      tree_inorder(tree_node_t *rootp)
      {
            if (rootp != NULL) {
                  tree_inorder(rootp->leftp);
                  printf("%d\n", rootp->key);
                  tree_inorder(rootp->rightp);
            }
      }
```

Chapter 15 Multiprocessing Using Processes and Threads

Section 15.1

1. A running program can be pre-empted at any time by the hardware interrupt system allowing access to the CPU in a predictable way that is independent of the programs that are running and adjustable based on criteria such as priority.
2. Linear programming involves writing a sequence of program instructions with each program instruction dependent upon the completion of the previous program instruction. Concurrent or parallel programming involves writing sets of program instructions that are independent of one another where each set of program instructions does not rely upon the completion of the other sets of program instructions.
3. Time sharing refers to allocating each system user a portion of the available CPU time thus sharing the CPU time among multiple users. A time slice is the unit of time allocated to a system user during their portion of the available CPU time.

Section 15.2

1. With the fork function.
2. With the wait function.
3. With the execl function.
4. The fork function creates a child process. The fork function assigns the process id of the child process to the variable pid in the parent process and 0 to the variable pid in the child process. The parent and child processes both call the printf function and display the value of the variable pid to the standard output.

Programming

1.
```
#include <stdio.h>
#include <unistd.h>
...
pid_t pid;
...
pid = fork();
printf("Process pid %d\n", pid);
...
if (pid < 0)
    printf("Error Creating Child Process\n");
else if (pid == 0)
    printf(" Child Process\n");
else
    printf("Parent Process\n");
```

Section 15.3

1. Pipes may only be used with processes that are running on the same CPU and that have a common ancestor.
2. With the pipe function.
3. With the dup2 function.

Section 15.4

1. With the pthread_create function.
2. With the pthread_mutex_init function.
3. With the pthread_mutex_lock and pthread_mutex_unlock functions.
4. Thread deadlock is when a thread is blocked waiting for a mutex that never gets released. As a result, the thread will remain blocked until the process ends.

Chapter 16 On to C++

Section 16.1

2.
```
cout << setw(5) << a;
cout << setiosflags( ios::fixed | ios::showpoint ) << setprecision(2)
     << setw(11) << b;
cout << setprecision(1) << setw(9) << c << "\n";
```

Programming

1.
```
//
// Monitor gasoline supply in storage tank.  Issue warning when supply
// falls below MIN_PCT % of tank capacity.
//

#include <iostream>
#include <iomanip>
```

```
      using namespace std;

      const double CAPACITY = 80000.0;  // number of barrels tank can hold
      const int MIN_PCT = 10;           // warn when supply falls below this
                                        //  percent of capacity
      const double GALS_PER_BRL = 42.0; // number of U.S. gallons in one barrel

      double monitor_gas(double min_supply, double start_supply);

      int
      main()
      {
         double start_supply, // input - initial supply in barrels
                min_supply,   // minimum number of barrels left without warning
                current;      // output - current supply in barrels

         // Compute minimum supply without warning
         min_supply = MIN_PCT / 100.0 * CAPACITY;

         // Get initial supply
         cout << "Number of barrels currently in tank> ";
         cin >> start_supply;

         // Subtract amounts removed and display amount remaining as long as
         // minimum supply remains.
         current = monitor_gas(min_supply, start_supply);

         // Issue warning
         cout << "Only " << setiosflags(ios::showpoint | ios::fixed) <<
            setprecision(2) << current << " barrels are left.\n\n";
         cout << "*** WARNING ***\n";
         cout << "Available supply is less than " << MIN_PCT << " percent of "
            << "tank's\n" << CAPACITY << "-barrel capacity.\n";

         return (0);
      }

      //
      // Computes and displays amount of gas remaining after each delivery
      // Pre: min_supply and start_supply are defined.
      // Post: Returns the supply available (in barrels) after all permitted
      //       removals.  The value returned is the first supply amount that is
      //       less than min_supply.
      //
      double
      monitor_gas(double min_supply, double start_supply)
      {
         double remov_gals,   // input - amount of current delivery in
                remov_brls,   //         in barrels and gallons
                current;      // output - current supply in barrels

         for  (current = start_supply;
               current >= min_supply;
               current -= remov_brls) {
             cout << setiosflags(ios::showpoint | ios::fixed) << setprecision(2)
                << current << " barrels are available.\n\n";
             cout << "Enter number of gallons removed> ";
             cin >> remov_gals;
             remov_brls = remov_gals / GALS_PER_BRL;

             cout << "After removal of " << remov_gals << " gallons (" <<
                remov_brls << " barrels),\n";
         }

         return (current);
      }
```

2.
```
      //
      //  function order by ordering three numbers
      //
      #include <iostream>
      #include <iomanip>
      using namespace std;

      void order(double& smp, double& lgp);
```

```
int
main()
{
   double num1, num2, num3;

   // Gets test data
   cout << "Enter three numbers separated by blanks> ";
   cin >> num1 >> num2 >> num3;

   // Orders three numbers
   order(num1, num2);
   order(num1, num3);
   order(num2, num3);

   // Displays results
   cout << "The numbers in ascending order are: " <<
      setiosflags(ios::showpoint | ios::fixed) << setprecision(2) <<
      num1 << " " << num2 << " " << num3 << "\n";

   return (0);
}

//
// Arranges arguments in ascending order.
// Pre: sm and lg are references to defined type double variables
// Post: variable referenced by sm contains the smaller of the type
//       double values; variable referenced by lg contains the larger
//
void
order(double& sm, double& lg)  // input/output
{
   double temp;  // temporary variable to hold one number during swap

   // Compares values referenced by sm and lg and switches if necessary
   if (sm > lg) {
      temp = sm;
      sm = lg;
      lg = temp;
   }
}
```

Section 16.2

2.
```
#include <iostream>
#include <cstdlib>
using namespace std;

class Ratio   {

public:
 Ratio ()  {}        // Default constructor
 Ratio ( int, int ); // Constructor  2 - components initialized
 void reduce ();     // reduces fraction

private:
 int   num;   // numerator
 int   denom; // denomintor

friend  istream& operator>>  ( istream&, Ratio& ) ;
friend  ostream& operator<<  (  ostream&, const Ratio& );
};

//
//  Constructor that initializes components
//
Ratio  ::  Ratio ( int numerator, int denominator )
{
   num = numerator;
   denom = denominator;
}

//
//  Reduces fraction represented by a Ratio object by
//    dividing num and denom by greatest common divisor
//
void Ratio ::  reduce ()   // NOT a const function
```

```
      {
        int n, m, r;
        n = abs(num);
        m = abs(denom);
        r = n % m;
        while (r != 0)  {
            n = m;
            m = r;
            r = n % m;
        }
        num  /=  m;
        denom  /= m;
      }

      //
      //   Extract from input source the two components of a Ratio
      //
      istream& operator>> (istream& is,  Ratio& oneRatio )
      {
         is  >> oneRatio.num  >> oneRatio.denom;
         return is;
      }

      //
      //  Display a Ratio object as a common fraction
      //
      ostream& operator<<  ( ostream& os, Ratio oneRatio )
      {
         os  <<  oneRatio.num;
         if (oneRatio.denom != 1)
            os  <<  " / "  <<  oneRatio.denom;
         return os;
      }

      //
      //  Driver to declare and manipulate a Ratio object
      //
      int
      main ()
      {
         Ratio aRatio;
         cout  <<
            "Enter numerator and denominator of a common fraction"
            << endl  <<  ">>> ";
         cin  >>  aRatio;
         cout  <<  endl  <<  "Fraction entered  = "  <<  aRatio
            <<  endl;
         aRatio.reduce();
         cout  <<  "Reduced fraction = " <<  aRatio  << endl;
         return 0;
      }
```

Programming

```
1.    class Can   {

      public:
         Can ()  {}        // Default constructor
         Can ( double, double, double ); // Constructor  2 -
                                   //   components initialized
         int capacity( double volume ) const;

      private:
         double empty_weight;   // grams
         double base_radius, height;  // centimeters

      friend istream& operator>> (istream& is, Can& acan);
      friend ostream& operator<< (ostream& os, Can acan);

      };
```

Part III
Answers to Review Questions

Chapter 1 Overview of Computers and Programming

1. Three kinds of information stored in a computer are characters, numbers, and program instructions.

2. Two functions of the CPU are coordinating all computer operations and performing arithmetic and logical operations on data.

3. Input devices: keyboard and mouse.
   Output devices: monitor and printer.
   Secondary storage devices: disk drive and CD-ROM drive.

4. Three categories of programming languages are machine languages, assembly languages, and high-level languages. Machine language is a computer's native language with instructions that are binary numbers (a series of 0's and 1's). In assembly language, computer operations are represented by mnemonics rather than binary numbers. High-level languages are programming languages whose instructions resemble everyday English.

5. A syntax error is an error in the grammatical form of an instruction line in a high-level language program.

6. A high-level language source program must be translated into machine language through compilation. The machine-language object program created by the compiler must be linked, possibly with other object files, producing a machine-language program ready for execution.

7. A bit is a binary digit, a 0 or a 1, and is the smallest element a computer can deal with. A byte is the amount of space required to store a single character. Generally there are eight bits to a byte. A memory cell is a grouping of bytes, the actual number varying from computer to computer.

8. C was originally used for writing system software, Pascal for teaching students to program in a careful, disciplined way, and FORTRAN for engineering and scientific applications.

9. RAM (random access memory) temporarily stores program and data in main memory, whereas ROM (read-only memory) stores programs or data permanently. The computer can retrieve or read, but cannot store or write information in ROM as you can in RAM.

10. The World Wide Web is the universe of Internet-accessible resources that are navigable through the use of graphical user interfaces.

11. Copy the applications programs from the purchased CD to the computer's hard disk.

12. (1) Digital Subscriber Line (DSL), a high-speed connection that uses the wires of a telephone line without interfering with simultaneous voice communication; (2) cable Internet access, two-way high-speed transmission of Internet data through two of the hundreds of channels available over the coaxial cable that carries cable television signals.

Chapter 2 Overview of C

1. The programmer's name, the date the current version was written, and a brief description of what the program does.

2. The correct variables are income, c3po, and item.

3. The value of the constant PI cannot be changed by the program.

4. G and MAX_SPEED.

5. Data Requirements

```
      Problem Constant
            CUTTING_RATE  2   /* cutting rate in feet per second  */
      Problem Inputs
            double ylength, ywidth; /* Length and width of yard   */
            double hlength, hwidth; /* Length and width of house  */
      Problem Output
```

```
                double cut_time;  /* time it takes to mow the lawn    */
          Relevant Formulas
                area of rectangle = length x width
                cutting time = (area of yard - area of house) / cutting rate
    Algorithm
    1.    Scan the rectangular yard dimensions.
    2.    Compute area of yard.
    3.    Scan the rectangular house dimensions.
    4.    Compute area of house.
    5.    Compute grass area by subtracting house area from yard area.
    6.    Compute cutting time by dividing grass area by cutting rate.
```

6. `printf("The average pH of citrus fruits is %.1f.\n", avg_citrus_pH);`

7. Three standard types of C are: int, double, and char.

8.
```
scanf("%c%c", &c1, &c2);
printf("\nThe two characters are %c and %c.\n", c1, c2);
scanf("%d%d%d", &n, &m, &p);
printf("The three integers are %d, %d, and %d.\n", n, m, p);
```

9. Algorithm:
```
1.    Get an integer value (invalue).
2.    Assign invalue * 2 - 10 to outvalue.
3.    Display result (outvalue).
```

Chapter 3 Top-Down Design with Functions

1. Top-down design is a problem solving method in which you first break a problem up into its major subproblems and then solve the subproblems to derive the solution to the original problem. A structure chart is a documentation tool that shows the relationships among the subproblems of a problem.

2. A function prototype is a declaration that tells the C compiler the data type of the function, the function name, and information about the arguments that the function expects.

3. A function is executed whenever it is called by the main function, or by other functions. A function prototype should appear after the preprocessor directives but before the main function. A function definition follows the main function. The order of the definitions does not matter, only the order in which they are called.

4. a. Top-down design: Functions facilitate organizing a solution with the top-down problem-solving method by allowing each function to solve one specific subproblem.
   b. Modular programming: Functions allow us to implement a program in logically independent sections, which make the program easier to understand and debug. For a team of programmers working together on a large program, the work can be divided more easily by having each programmer be responsible for a particular set of functions.
   c. Code reuse: Once written and tested, functions can be executed many times within the same program, as well as being used as the building blocks for new programs.

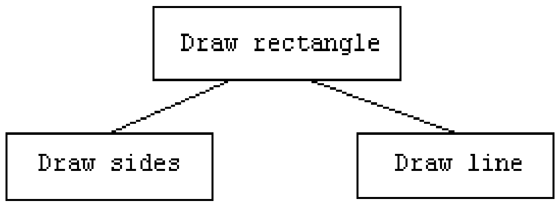
5. The use of functions is a more efficient use of the programmer's time. A program with functions usually has fewer lines of code than an equivalent program without functions. Since there is less code to write and debug, it takes a programmer less time to write a program with functions. It also allows the programmer to approach the problem using the highly efficient top-down design method.

   A program with functions executes a bit slower than an equivalent program without functions. This is because there is some overhead in calling the function and returning back from the function.

6.
```
/*  This program computes the hypotenuse of a right triangle given two legs. */

#include <stdio.h>
#include <math.h>

int
main(void)
{
  double leg1, leg2, hypotenuse;

  printf("\nType in the lengths of the two legs of a right triangle: ");
  scanf("%lf%lf", &leg1, &leg2);
  hypotenuse = sqrt(pow(leg1, 2.0) + pow(leg2, 2.0));
  printf("\nThe hypotenuse is %.2f.\n", hypotenuse);
  return (0);
}
```

7.
```
/*  This program draws double border rectangles made of asterisks. */

#include <stdio.h>
#include <math.h>

void draw_sides(void);
void draw_line(void);

int
main(void)
{
   draw_line();
   draw_line();
   draw_sides();
   draw_line();
   draw_line();

   return (0);
}

void
draw_line(void)
{
   printf("************\n");
}

void
draw_sides(void)
{
   printf("**        **\n");
   printf("**        **\n");
   printf("**        **\n");
   printf("**        **\n");
}
```

8.



9. `int script(int num_spaces, char char_to_display, int num_of_chars);`

Chapter 4 Selection Structures: if and switch Statements

1. if

2. Function ice_forming will be called.

3.
```
if (grade < 0)
      printf("Bad data.\n");
else if (grade == 0)
      printf("None.\n");
else if (grade < 6)
      printf("1-5 Elementary School.\n");
else if (grade < 9)
      printf("6-8 Middle School.\n");
else if (grade < 13)
      printf("9-12 High School.\n");
else
      printf(">12 College.\n");
```

4.
```
      switch (inventory){
        case 'B':
        case 'C':
            total_paper = total_paper + paper_order;
            break;

        case 'E':
        case 'F':
        case 'D':
            total_ribbon = total_ribbon + ribbon_order;
            break;

        case 'A':
        case 'X':
            total_label = total_label + label_order;
            break;

        case 'M':
            break;

        default:
            printf("Invalid inventory code!\n");
      }
```

5.
```
      if (weight >= opt_min  &&  weight <= opt_max  &&
            age >= age_min  &&  age <= age_max  && !smoker)
                  printf("Acceptable astronaut candidate.\n");
```

6.
```
      if (age > 59)
            if ( sts == 'W')
                  printf("Working senior\n");
            else
                  printf("Retired senior\n");
      else if (age > 20)
            printf("Adult\n");
      else if (age > 12)
            printf("Teen\n");
      else
            printf("Child\n");
```

Chapter 5 Repetition and Loop Statements

1. The control-variable initialization and control-variable update sections of the sentinel-controlled loop and the endfile-controlled loop are alike in that they both scan in the next data value. They are different in that the loop repetition condition of the sentinel-controlled loop checks the last data item scanned to see if it matches the sentinel value, while the endfile-controlled loop checks the return value of the (f)scanf to see if the end of the file has been reached.

2.
```
      /* Display a list of Celsius temperatures and their sum. */

      #include <stdio.h>
      #define SENTINEL -275

      int
      main(void)
      {
            int sum = 0, temp;

            printf("\nInput the first temperature (or %d to quit)> ",SENTINEL);
            scanf("%d", &temp);
            while (temp != SENTINEL) {
                  printf("Temperature entered: %d\n", temp);
                  sum += temp;
                  printf("\nEnter the next temperature (or %d to quit)> ",SENTINEL);
                  scanf("%d", &temp);
            }
            printf("\nThe sum of the temperatures is %d.\n", sum);
            return (0);
      }
```

```
3.                                                                     y2   y1   x2   x1
      printf("\nEnter four numbers> ");
      scanf("%lf%lf%lf%lf",&y2,&y1,&x2,&x1);    --->                   4    2    8    4

      for (slope = (y2 - y1) / (x2 - x1); --->             slope = .5
      slope != SPECIAL_SLOPE; --->              True
      printf("Slope is %5.2f.\n", slope); --->             Slope is  0.50.

      scanf("%lf%lf%lf%lf",&y2,&y1,&x2,&x1);    --->                   1    4    2    1
      slope = (y2 - y1) / (x2 - x1);      --->             slope = -3.0
      slope != SPECIAL_SLOPE; --->              True
      printf("Slope is %5.2f.\n", slope); --->             Slope is -3.00.

      scanf("%lf%lf%lf%lf",&y2,&y1,&x2,&x1);    --->                   9    3    3    1
      slope = (y2 - y1) / (x2 - x1);      --->             slope = 3.0
      slope != SPECIAL_SLOPE; --->              True
      printf("Slope is %5.2f.\n", slope); --->             Slope is 3.00.

      scanf("%lf%lf%lf%lf",&y2,&y1,&x2,&x1);    --->                  -22   10    8    2
      slope = (y2 - y1) / (x2 - x1);      --->             slope = -5.33
      slope != SPECIAL_SLOPE; --->              True
      printf("Slope is %5.2f.\n", slope); --->             Slope is -5.33.

      scanf("%lf%lf%lf%lf",&y2,&y1,&x2,&x1);    --->                   3    3    4    5
      slope = (y2 - y1) / (x2 - x1);      --->             slope = 0
      slope != SPECIAL_SLOPE; --->              False

      return (0);

4.    #include <stdio.h>
      #define SPECIAL_SLOPE  0.0

      int
      main(void)
      {
            double slope, y2, y1, x2, x1;

            printf("\nEnter four numbers separated by spaces.");
            printf("\nThe last two numbers cannot be the");
            printf(" same, but\nthe program terminates if ");
            printf("the first two are. ");

            printf("\nEnter four numbers> ");
            scanf("%lf%lf%lf%lf",&y2,&y1,&x2,&x1);
            slope = (y2 - y1)/(x2 - x1);

            while (slope != SPECIAL_SLOPE){
                  printf("Slope is %5.2f.\n",slope);
                  printf("Enter four more numbers> ");
                  scanf("%lf%lf%lf%lf",&y2,&y1,&x2,&x1);
                  slope = (y2 - y1)/(x2 - x1);
            }

            return (0);
      }

5.    count = 0;
      for (i = 0;  i < n;  ++i) {
            scanf("%d", &x);
            if (x == i)
                  ++count;
      }

6.    for (i = n;  i < max;  ++i)

7.    /* Get data from user until in_val is in the range 0 - 15. */
      do {
            /* No errors detected yet. */
            error = 0;

            /* Get a number from the user. */
            printf("Enter an integer in the range from 0 through");
            printf(" 15 inclusive> ");

            /* Validate the number. */
            if (scanf("%d", &in_val) != 1) {  /* in_val didn't get a number */
```

```
                  error = 1;
                  scanf("%c", &skip_ch);
                  printf("Invalid character <<%c>>. ", skip_ch);
                  printf("Skipping rest of line.\n");
            } else if (in_val < 0 || in_val > 15) {
                  error = 1;
                  printf("Number %d is not in range.\n", in_val);
            }

            /* Skip rest of data line. */
            do
                  scanf("%c", &skip_ch);
            while (skip_ch != '\n');
      } while (error);
```

Chapter 6 Modular Programming

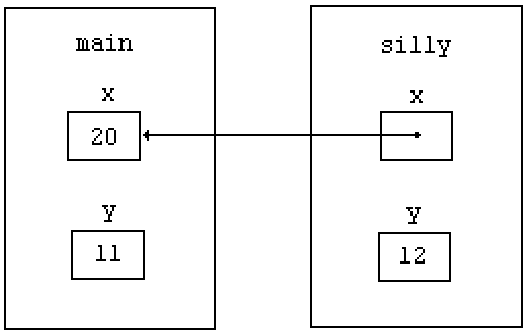
1.
```
      /*
       *  Returns letter grade based on points
       *  90 - 100 = A;    80 - 89 = B;    70 - 79 = C;    60 - 69 = D;    Below 60 = F
       */
      char
      letter_grade(int points)
      {
         char grade;
         if (points < 60)
            grade =  'F';
         else if (points < 70)
            grade = 'D';
         else if (points < 80)
            grade = 'C';
         else if (points < 90)
            grade = 'B';
         else
            grade = 'A';

         return (grade);
      }
```

2. A function that produces a single result and passes that result back by a return statement can be called anywhere the function's result is needed.

3. When a function is called, memory is allocated in the function data area for each of the function's formal parameters and all its local variables. The value of each actual parameter is stored in the memory cell allocated to its corresponding formal input parameter. The function data area for a formal output parameter contains the address of or a pointer to the variable used as its actual parameter. This allows the value of the actual parameter to be changed by the function.

4. Only main and grumpy can call grumpy because the other two functions have their own conflicting declarations of grumpy.

5.



6. a. Programs that use functions typically have fewer lines. Since functions can be called more than once during a program, the number of lines in a program has been reduced as soon as a second call to a function is needed.

b. Subprograms actually reduce the errors in a program since a function can be developed and tested independently. When functions are not used, logic errors and syntax errors occur repeatedly as the function's lines are included every time the function result is needed.

Chapter 7 Simple Data Types

1. The data type int uses less storage and stores an integer exactly. Operations using data type int run faster. The data type double allows a larger range of numbers to be represented, and these numbers can have fractional parts.

2. Cancellation occurs when a large number added to a small number simply results in the large number.

   Arithmetic overflow occurs when the result of a computation is a number that is too large to be represented.

   Arithmetic underflow occurs when an operation produces a result too small to be represented.

3.
```
int
is_vowel(char ch)
{
  int ans;

  switch (ch) {
  case 'A': case 'a':
  case 'E': case 'e':
  case 'I': case 'i':
  case 'O': case 'o':
  case 'U': case 'u':   ans = 1;
                        break;

  default:              ans = 0;
  }
  return (ans);
}
```

4.
```
double
series (int n,
        double x)
{
   double sum = 0.0, t, term = 1.0;

   for (t = 1.0;  t <= n;  ++t) {
      term *= x / t;
      sum += term;
   }

   return (sum);
}
```

5.
```
#include <stdio.h>
#include <stdlib.h>

typedef enum
      {winter, spring, summer, fall}
season_t;

void
print_season(season_t season)
{
      switch (season) {
      case winter: printf("Winter\n");
                   break;

      case spring: printf("Spring\n");
                   break;

      case summer: printf("Summer\n");
                   break;
```

```
            case fall  : printf("Fall\n");
                         break;
            }
      }
```

6.
```
      typedef enum
            {july, august, september, october, november, december,
             january, february, march, april, may, june}
      fiscal_t;

      fiscal_t month;

      switch (month) {
            case june: case july:
            case august:      printf("summer\n");
                              break;

            case september: case october:
            case november:    printf("fall\n");
                              break;

            case december: case january:
            case february:    printf("winter\n");
                              break;

            case march: case april:
            case may:         printf("spring\n");
                              break;

            default:          printf("invalid month\n");
      }
```

7.
```
      for (i=1;  i <= 10;  ++i)
         printf("%5.1f", i * 0.1);
```

8. c

9. cast

Chapter 8 Arrays

1. The loop counter goes from 0 to 8, whereas the array x's subscripts range from 0 to 7. When x[8] is set to 8, it is possible that the program will abort, but it is more likely that the program will simply produce incorrect results.

2. double exper[7];

3. True.

4. Randomly or sequentially.

5.
```
      max = 0;
      min = 0;
      for (t = 1; t < 20; ++t)  {
         if (x[t] > x[max]) max = t;
         if (x[t] < x[min]) min = t;
      }

      printf("Max: %d  Min: %d", max, min);
```

6.
```
      /*
       *  Stores in y the first n elements of x in reverse order.
       */
      void
      reverse(int       y[], /* output array  */
              const int x[], /* input array */
              int       n)   /* number of elements */
      {
         int t;

         for (t = 0;  t < n;  ++t)
           y[t] = x[n - t - 1];
      }
```

7.
```
for (t = 0;  t < 5;  ++t) {
   sum = 0.0;
   for (t1 = 0;  t1 < 3;  ++t1)
     sum += table[t][t1];
   printf("Row %d sum is %.2f.\n", t + 1, sum);
}
```

Five row sums are displayed.
Each sum includes three elements.

8.
```
for (t = 0;  t < 3;  ++t) {
   sum = 0.0;
   for (t1 = 0;  t1 < 5;  ++t1)
     sum += table[t1][t];
   printf("Column %d sum is %.2f.\n", t + 1, sum);
}
```

Three column sums are displayed.
Each sum includes five elements.

Chapter 9 Strings

1. s5 = "Satu"

2. s10 = "und"

3. 8

4. s20 = "SundaySaturday"

5.
```
/*
 *  Adds blanks to the end of s until s is its full declared size.
 */
char *
blank_pad(char *s,    /* input/output string */
          int   size) /* array size          */
{
   int t;
   for (t = strlen(s);  t < size - 1;  ++t)
      s[t] = ' ';
   s[size-1] = '\0';

   return (s);
}
```

6.
```
/*
 * Returns a copy of strin with first occurrence of c1 deleted.
 */
char *
del_char(char       *strout,  /* output string  */
         const char *strin,   /* input string   */
         char        c1,      /* char to delete */
         int         maxsize) /* max size of output */
{
   char temp[STRSIZ];
   int c1pos = 0, sin_cnt;

   while (sin_cnt < strlen(strin) && strin[c1pos] != c1)
      ++c1pos;
   if (strin[c1pos] == c1) {
      strncpy(strout, strin, c1pos);  strout[c1pos] = '\0';
      strcat(strout, &strin[c1pos + 1]);
   }

   return (strout);
}
```

7.
```
#define FALSE 0
#define TRUE  1

int
isvowel(int ch)
{
   int ret_val = FALSE;

   if (isalpha(ch)) {
      if (islower(ch))
         ch = toupper(ch);
      switch (ch) {
      case 'A':
      case 'E':
      case 'I':
      case 'O':
      case 'U':
         ret_val = TRUE;
      }
   }

   return (ret_val);
}

int
isconsonant(int ch)
{

   int ret_val = FALSE;

   ret_val = isalpha(ch) && !isvowel(ch);

   return (ret_val);
}
```

8. b

9. d

10. a

Chapter 10 Recursion

1. When a difficult problem can be defined in terms of a simpler version of itself, much of the complexity of the problem is automatically handled by recursion. The algorithm developer is then free to nibble at the problem one simplifying step at a time.

2. In most cases, if there are well-designed recursive and iterative solutions to the same problem, the recursive solution will require more time and space because of the extra function calls.

3. A terminating condition is a condition that is true when recursion can stop and a simple case can be solved. A simple case is a case of a problem that has an answer that can be directly expressed without using recursion.

4.
```
/*
 * Computes the sum of the first num_elements values in array nums.
 */
double
sum_array(const double nums[],
          int          num_elements)
{
   double sum;

   if (num_elements == 1)
      sum = nums[0];
   else
      sum = nums[0] + sum_array(&nums[1], num_elements - 1);

   return (sum);
}
```

5.
```
/*
 * Counts the number of vowels in string.
 */
int
count_vowels(const char *string)
{
   int ans;
```

```
      if (strlen(string) == 0)
         ans = 0;
      else if (is_element(string[0], "AaEeIiOoUu"))
         ans = 1 + count_vowels(&string[1]);
      else
         ans = count_vowels(&string[1]);

      return (ans);
   }
```

6.
```
   #define TOLERANCE .001

   /*
    * Determines if remaining input forms a geometric sequence.
    */
   void
   check_geometric(double ratio,
                   double lastterm)
   {
      double term;
      int    status;

      printf("\n\nEnter next term: ");
      status = scanf("%lf", &term);

      if (status != 1)
         printf("List forms a geometric sequence.\n");
      else if (fabs(term / lastterm - ratio) <= TOLERANCE)
         check_geometric(ratio, term);
      else
         printf("Input halted at %.2f.  List does not form a geometric sequence.\n", term);
   }
```

7.
```
   /*
    *  Returns the array index of the last non-blank character in the input string.
    *  Post:  string value is modified
    */
   int
   find_last(char string[])
   {
      int len, ans;

      len = strlen(string);

      if (string[len - 1] != ' ') {
         ans = len - 1;
      } else {
         string[len - 1] = '\0';
         ans = find_last(string);
      }

      return (ans);
   }
```

Chapter 11 Structure and Union Types

1.
```
   typedef struct {
         char   name[20],
         street_address[30];
         double monthly_bill;
   } subscriber_t;
```

2.
```
   int
   main(void)
   {
      olympic_t competition;

      printf("\nEnter event:   ");
      gets(competition.event);
      printf("Enter entrant: ");
      gets(competition.entrant);
      printf("Enter country: ");
      gets(competition.country);
      printf("Enter place:   ");
      scanf("%d", &competition.place);
```

```
         printf("\nEvent:   %s\n", competition.event);
         printf("Entrant: %s\n", competition.entrant);
         printf("Country: %s\n", competition.country);
         printf("Place:   %d\n", competition.place);

         return (0);
      }
```

3.
```
      scan_olympic(&competition);
```

4.
```
      typedef struct {
         char   name[15],
                start_date[15];
         double hrs_worked;
      } summer_help_t;

      /* Prototype for scan_sum_hlp */

      int
      main(void)
      {
            summer_t operator;   /*  summer_t, not struct */

            scan_sum_hlp(&operator);   /*  & needed:  operator is an output argument */
            printf("Name:  %s\nStarting date:  %s\nHours worked:  %.2f\n", operator.name,
                    operator.start_date, operator.hrs_worked);   /*  Each component must be
                                                                     listed separately  */

            return (0);
      }

      /* code for function scan_sum_hlp goes here */
```

5.
```
      typedef struct {
              char street_address[30],
                   city[20],
                   state[3],
                   zip[11];
      } address_t;

      typedef struct {
              char description[50],
                   time[10],
                   days[6];
      } schedule_t;

      typedef struct {
              double     gpa;
              char       major[20];
              address_t  address;
              int        num_classes;
              schedule_t class_schedule[6];
      } student_t;
```

Chapter 12 Text and Binary File Processing

1. Files are generally stored on a floppy disk, a hard disk, or magnetic tape.

2.
```
      /* Make backup copy one character at a time  */
      for  (ch = getc(inp);  ch != EOF;  ch = getc(inp)){
            putchar(ch);
            putc(ch, outp);
      }
```

3.
```
      #include <stdio.h>

      #define NAME_LEN 21
      #define SOCSEC_LEN 12

      typedef struct {
              char   name[NAME_LEN];
              char   socsec[SOCSEC_LEN];
              double gross, taxes, net;
      } emppay_t;


      int
      main(void)
      {
         FILE *fpin, *fpout, *fpbin;
         emppay_t emp;
         int status;

         if ((fpin = fopen("empstat.txt", "r")) == NULL)
            printf("Unable to open empstat.txt.\n");
         else if ((fpout = fopen("report.txt", "w")) == NULL)
            printf("Unable to open report.txt.\n");
         else if ((fpbin = fopen("empstat.bin", "w")) == NULL)
            printf("Unable to open empstat.bin.\n");
         else {
            fprintf(fpout, "NAME                SOC.SEC.NUM     GROSS     TAXES     NET\n\n");
            for (status = fscanf(fpin, "%s%s%lf%lf%lf", emp.name, emp.socsec, &emp.gross,
                                 &emp.taxes, &emp.net);
                 status == 5;
                 status = fscanf(fpin, "%s%s%lf%lf%lf", emp.name, emp.socsec, &emp.gross,
                                 &emp.taxes, &emp.net)) {
               fprintf(fpout, "%-20s%-lls  %8.2f  %8.2f  %8.2f\n", emp.name, emp.socsec,
                       emp.gross, emp.taxes, emp.net);
               fwrite(&emp, sizeof (emppay_t), 1, fpbin);
            }
         }
         if (fpin != NULL) fclose(fpin);
         if (fpout != NULL) fclose(fpout);
         if (fpbin != NULL) fclose(fpbin);

         return (0);
      }
```

4. Binary files consist of components that are copied directly from computer memory without any transformation. Therefore, they tend to be much more compact than text files, and they can be read and written much faster. The major problem with binary files is that they may not be readable by the same program compiled and executed on a different computer, and they cannot be created or proofread using a text editor.

5.
```
      #include <stdio.h>

      #define NAME_LEN 21
      #define SOCSEC_LEN 12
      typedef struct {
              char   name[NAME_LEN];
              char   socsec[SOCSEC_LEN];
              double gross, taxes, net;
      } emppay_t;

      typedef struct {
              char   socsec[SOCSEC_LEN];
              double gross;
      } socgros_t;

      int
      main(void)
      {
         FILE *fpin, *fpout, *fpbin;
         emppay_t emp;
         socgros_t outval;
         int status;

         if ((fpin = fopen("empstat.bin", "r")) == NULL)
            printf("Unable to open empstat.bin.\n");
         else if ((fpout = fopen("ssngross.bin", "w")) == NULL)
            printf("Unable to open ssngross.bin.\n");
         else
            for (status = fread(&emp, sizeof (emppay_t), 1, fpin);
                 status == 1;
                 status = fread(&emp, sizeof (emppay_t), 1, fpin)) {
               strcpy(outval.socsec, emp.socsec);
               outval.gross = emp.gross;
               fwrite(&outval, sizeof (socgros_t), 1, fpout);
            }
         if (fpin != NULL) fclose(fpin);
         if (fpout != NULL) fclose(fpout);

         return (0);
      }
```

6. A file pointer is the address of a structure containing the information necessary to access an open input or output file.

7.

```
      typedef struct {
              int row, col, value;
      } mat_val_t;

      /*
       * Stores sparse matrix nums as a binary file of mat_val_t structures.
       */
      void
      store_sparse(FILE *fp,            /* pointer to open binary output file */
                   int nums[][MAXCOL])  /* input - sparse matrix              */
      {
         int r, c;
         mat_val_t one_entry;

         for (r=0;  r < MAXROW;  ++r)
            for (c=0;  c < MAXCOL;  ++c)
               if (nums[r][c] != 0) {
                  one_entry.row = r;
                  one_entry.col = c;
                  one_entry.value = nums[r][c];
                  fwrite(&one_entry, sizeof (mat_val_t), 1, fp);
               }
      }
```

8. There would be no difference in the function prototypes, which means that the function interface comment must indicate the file type to be written.

Chapter 13 Programming in the Large

1. Procedural abstraction is the philosophy that function development should separate the concern of what is to be achieved by a function from the details of how it is to be achieved.

   Data abstraction is the specification of data objects and their operators without concern for specific implementation.

2. C allows the creation of personal libraries that can contain the needed functions and data structures to encapsulate data objects and their operators.

3. A header file contains the constants, data structures, prototypes, and comments necessary for a programmer or a program to use a library. The implementation file contains the actual code needed to implement the library. The header file defines the interface between a library and a program.

4. An include file in the system directory is referenced by <filename>; one in the program's directory is referenced by "filename".

5. You can be sure the function call mac(++a, b) will be valid, because ++a is evaluated just once, on entry to the function. In contrast, ++a is evaluated each time it is substituted in MAC.

6. Parentheses should be included around each use of a parameter and around the entire expression.

7. The five storage classes are auto, extern, static, register, and typedef.
   Variables declared at the top level are of storage class extern.
   Function parameters are of storage class auto.
   Local variables are of storage class auto.

8. The storage class register allows a programmer to indicate that a variable will be used often (like a loop counter) and should be stored in a high speed memory cell inside the CPU if at all possible.

9. Declaring the array as a global array tends to defeat the purpose of procedural and data abstraction, since all modules must now take care to use the right array name or the functions will not work. Moreover, other functions could also manipulate the array, making debugging the program difficult.

10. The exit function is typically used to terminate a program after an error has occurred. A return value of 1 indicates an error.

11. The defined operator is used in #if preprocessor directives to coordinate conditional compilation of parts of a C program.

12. The value of its first parameter is never less than 1 because the program name is considered the first argument.

Chapter 14 Dynamic Data Structures

1. Dynamic data structures are allocated on the heap as a result of executing calls to memory allocation functions such as malloc and calloc. These structures can grow and shrink as a program executes, and they are referenced through pointers. They remain allocated until explicitly deallocated by a call to free. Non-dynamic data structures are allocated on the stack when the function to which they belong is called and are deallocated when the function returns. Their size is known when the program is compiled--they do not grow and shrink. They are referenced by their names.

2. A linked-list is made up of nodes. Each node can be split into two parts, the data and the pointer to the next node. The pointer to the next node contains the address of the next node. Besides the nodes, a linked-list requires a pointer to the head of the list (the first node).

3.
```
typedef struct word_node_s {
   char word[10];
   struct word_node_s *next;
} word_node_t;

word_node_t *wp, *qp;
```

1st statement allocates a word_node_t structure, storing the pointer in wp.
2nd statement stores "ABC" in the word component of the new structure.
3rd statement allocates another word_node_t structure, storing the pointer in the "next" component of the first word_node_t structure allocated.
4th statement stores a copy of the pointer to the most recently allocated word_node_t structure in qp.
5th statement stores "abc" in the word component of the newest structure.
6th statement stores NULL in the "next" component of the newest structure.
End result is a two-element linked list storing the data "ABC" "abc".

4.
```
np = (name_node_t *)malloc(sizeof (name_node_t));
strcpy(np->name, "Kennedy");
np->restp = NULL;
qp = np;
np = (name_node_t *)malloc(sizeof (name_node_t));
strcpy(np->name, "Roosevelt");
np->restp = qp;
qp = np;
list.headp = (name_node_t *)malloc(sizeof (name_node_t));
strcpy(list.headp->name, "Washington");
list.headp->restp = qp;
list.size = 3;
```

5.
```
np = (name_node_t *)malloc(sizeof (name_node_t));
strcpy(np->name, "Eisenhower");
np->restp = list.headp->restp->restp;
list.headp->restp->restp = np;
++list.size;
```

6.
```
name_list_t
delete_last(name_list_t list)
{
   name_node_t *np, *op;
```

```
         if (list.size > 0) {
            if (list.size == 1) {
               free(list.headp);
               list.headp = NULL;
            } else {
               for (np = op = list.headp;  np->restp != NULL;  np = np->restp)
                  op = np;
               free(op->restp);
               op->restp = NULL;
            }
            --list.size;
         }
         return (list);
      }
```

7.
```
      name_list_t
      place_first(name_list_t list, char *name)
      {
         name_node_t *np;

         np = (name_node_t *)malloc(sizeof (name_node_t));
         strcpy(np->name, name);
         np->restp = list.headp;
         list.headp = np;
         ++list.size;

         return (list);
      }
```

8.
```
      name_node_t *
      new_list(name_node_t *old_listp)
      {
         name_node_t *newp;

         if (old_listp == NULL) {
            newp = NULL;
         } else {
            newp = (name_node_t *)malloc(sizeof (name_node_t));
            strcpy(newp->name, old_listp->name);
            newp->restp = new_list(old_listp->restp);
         }

         return (newp);
      }

      name_list_t
      copy_list(name_list_t list)
      {
         name_list_t listcopy;

         listcopy.size = list.size;
         listcopy.headp = new_list(list.headp);

         return (listcopy);
      }
```

9.
```
      name_list_t
      delete_smith(name_list_t list)
      {
         name_node_t *np,*op;

         if (list.size > 0) {
            while (strcmp(list.headp->name, "Smith") == 0) {
               np = list.headp;
               list.headp = np->restp;
               --list.size;
               free(np);
            }
         }

         if (list.size > 0) {
            op = list.headp;
            np = list.headp->restp;
            while (np != NULL) {
               if (strcmp(np->name, "Smith") == 0) {
                  op->restp = np->restp;
                  free(np);
                  --list.size;
                  np = op->restp;
               } else {
                  op = np;
                  np = np->restp;
               }
            }
         }

         return (list);
      }
```

10. A queue.

11. The nodes of a simple linked list include a single pointer field which points to the rest of the list. Each binary tree node contains two pointer fields--one pointing to the left subtree, the other pointing to the right subtree. When searching a linked list, one must look at each element in turn, beginning with the first. A search of a binary search tree begins with the root node, and then considers either the left subtree of the root node or the right subtree, depending on the relationship between the target key and the root key. At each node, one of the two subtrees can be ignored. Insertion in a linked list is simple if order is irrelevant. Insertion in an ordered linked list requires finding the node that precedes the correct insertion point, copying this node's pointer field into the pointer component of the newly allocated node, and linking the old node to the new one. Insertion in a binary search tree does not require such relinking, since new nodes are always inserted as leaf nodes.

13. A binary tree node is a leaf if both of its subtree pointers are null.

Chapter 15 Multiprocessing Using Processes and Threads

1. Pre-emptive multi-tasking utilizes a hardware interrupt system to interrupt the program that is currently running allowing other programs to gain access to the CPU in a way that is predictable, independent of which programs are running and adjustable based on criteria such as priority.

2. A single CPU can execute only one program instruction at a time and is therefore not able to execute multiple program instructions at the same time in parallel with each other.

3. In the linear programming model, a program instruction is dependent upon the completion of the previous program instruction. Each task must be completed before the next task can begin. Because each task is dependent upon the completion of the previous task a parallel problem must be modeled serially in order to use the linear programming model.

4. A new process, referred to as a child, is created by an existing process, referred to as a parent, by calling the fork function. The child process is created as a copy of the parent process with its own address space; that is, a separate area of memory for storing all data and information required for the process to execute.

5. A process can exit independently of its parent process or child processes. A child process can exit before or after its parent process. It is generally preferred to wait until all child processes have exited before exiting the parent process.

6. A process can replace its instructions and memory space with different instructions and a different memory space by calling the execl function. The original process instructions and memory space no longer exist.

7. Pipes are limited to processes on the same CPU with a common ancestor because the pipe variable is declared within the original process memory space and is only available to that process or any child process created by that process because all child processes get their own copy of the address space from the parent process.

8. Pipes are created with the pipe function that returns two file descriptors in an integer array, a read file descriptor and a write file descriptor. The pipe can be read from by calling the read function with the read file descriptor of the pipe integer array. The pipe can be written to by calling the write function with the write file descriptor of the pipe integer array.

9. A pipe can be connected to either standard input or standard output by calling the function dup2 which duplicates and assigns the pipe to standard input or standard output. This is necessary before the process replaces itself by calling the function execl because the new process instructions do not have access to the original process memory area where the pipe was declared.

10. A thread is created with the pthread_create function. The thread of control that created the new thread continues executing at the next program instruction. When the new thread becomes the thread of control it begins executing at the first program instruction in the function specified by the argument start_routine in the call to pthread_create.

11. Mutex locks are important because they are used to synchronize access to shared resources in order to ensure that all threads have access to consistent data values at all times.

12. A thread can lock a mutex that is currently not locked by another thread. Only one thread at a time can lock a mutex. Any thread that tries to lock a mutex that is already locked by another thread will be blocked until the mutex has been released by the other thread.

13. A thread can deadlock with itself if it has a mutex locked and then attempts to lock the same mutex again. A thread can deadlock with another thread if each thread tries to lock a mutex that the other thread already has locked.

Chapter 16 On to C++

1. An object is a semiautonomous agent that has prescribed responsibilities. The C++ class definition facility permits definition of a type of objects.

2. tree_1.grow(5)   (Any int can be used in place of 5.)
   grow(5)

3. To display integer values, set the field width by inserting setw(n) in the output stream just prior to the insertion of the integer to display. The integer will be right-justified in a field of n columns. To display floating point numbers, insert in the output stream setiosflags(ios::fixed, ios::showpoint). Then set the field width by inserting setw(n) in the stream. A call to setprecision inserted in the output stream will determine the number of digits diplayed to the right of the decimal point.

4. A friend function or operator may access the private members of an object.

5. C++ allows functions to use value and/or reference parameters. A value parameter is a copy of the corresponding argument, whereas a reference parameter is the address of the corresponding argument. A reference parameter lets the function/operator refer to the original copy of the argument.

6. Declaring a class member function to be constant makes a commitment that the function will not change the values of any of the calling object's data members.

7. The system determines which definition of the overloaded output operator to call based on the data type of the right operand.

8. Whether or not the data members of seedling will be initialized is determined by the definition of class Tree's default constructor (the constructor that takes no parameters).

Part IV
Selected Programming Project Solutions

```
/*****************************   CHAPTER 2  -  PROJECT 4   *****************************
 *        by Brent Youngers
 *  Convert a temperature in degrees Fahrenheit to degrees Celsius */
 */

#include <stdio.h>
#define DIFF  .555556

int
main(void)
{
 int fahrenheit;   /* temperature in degrees Fahranheit */
 double celsius;   /* temperature in degrees Celsius */

 printf("Enter a temperature in degrees Fahrenheit: ");
 scanf("%d", &fahrenheit);

 celsius = DIFF * (fahrenheit - 32);
 printf("%d degrees in Fahrenheit is %.2f degrees in Celsius", fahrenheit,
       celsius);

 return (0);
}


/*****************************   CHAPTER 3  -  PROJECT 4   *****************************
 *
 *  This program approximates n! using a formula proposed by R. W. Gosper
 */

#include <stdio.h>
#include <math.h>

#define PI 3.14159265

/* Prototype */
void instruct(void);

int
main(void)
{
  int n;

  double n_fact,        /* approximation of n! */
         part1, part2;  /* intermediate values in Gosper’s calculation */

/* Display instructions */
  instruct();

/* Get value for n */
  printf("n> ");
  scanf("%d", &n);

/* Approximate n! */
  part1 = 2 * n + 1.0/3.0;
  part2 = sqrt(part1 * PI);
  n_fact = pow(n, n) * exp(-n) * part2;

/* Display results */
  printf("2n + 1/3 = %.5f\n", part1);
  printf("sqrt((2n + 1/3)PI) = %.5f\n", part2);
  printf("\n%d! equals approximately %.5f\n", n, n_fact);

  scanf("%d", &n);
  return (0);
}

/*
 *  This function displays a set of user instructions.
 */
```

```
void
instruct(void)
{
  printf("This program approximates n! using a formula proposed\n");
  printf("by R.W. Gosper.\n");
  printf("\nWhen you are prompted for a value of n, enter an integer\n");
  printf("between 0 and 8\n\n");
}



/******************************
 *
 *  Cyclist Coasting Program
 *  Programmer:  John Regnier
 */

#include <stdio.h>      /*  printf, scanf definitions */

/* Prototype */
void instruct(void);

#define T (1/60.0)                  /*  Time Interval is 1 minute */

int
main(void)
{
      double Vf,  /*  Final Velocity      */
             Vi,  /*  Initial Velocity    */
              a,  /*  Acceleration  */
             Tf;  /*  Time for Vf=0 */

      instruct();

      /*  Obtain Initial Velocity   */
      printf("\n Enter Initial Velocity (m/hr.)>  ");
      scanf("%lf", &Vi);

      /*  Obtain Final Velocity     */
      printf(" Enter Final Velocity (m/hr.)>  ");
      scanf("%lf", &Vf);

      /*  Calculate Acceleration    */
      a = (Vf - Vi) / T;

      /*  Calculate Time when Vf=0  */
      Tf = (Vi / (-a));

      /*  Display Results           */
      printf("\n\n");
      printf("For Initial Velocity %.2f and Final Velocity %.2f ->", Vi, Vf);
      printf("\n  The uniform acceleration is %.2f miles/hr^2, and ", a);
      printf("\n  the time to stop is %.2f hours.\n\n", Tf);

      return (0);
}

/*
 *  Displays Instructions to the User
 */
void
instruct(void)
{
      printf("\n\n\n\n\n\n  *****  CYCLIST COASTING PROGRAM  *****\n");
      printf("This program will ask you for the following: \n");
      printf("  initial velocity    (Vi) miles/hr.\n");
      printf("  final velocity      (Vf) miles/hr.\n");
      printf("...Then it will tell you the uniform acceleration\n");
      printf("over the time interval of 1 minute.  Also it will\n");
      printf("tell you the total time that it takes for your\n");
      printf("bike to come to rest.\n");
      printf(" *****NOTE:  Program assumes Vf < Vi *****  \n");
}


/******************************   CHAPTER 4  -  PROJECT 4  *****************************
```

```
 *
 *  Paul Onakoya
 *
 *  This program reports the contents of a compressed-gas cylinder based on the first
 *  letter of the cylinder's color. The program gives an appropriate response to an
 *  unrecognized letter.
 */

#include <stdio.h>

int
main(void)
{

    char input;  /* First letter of cylinder's color */

    printf("Enter the first letter of the cylinder's color => ");
    scanf("%c", &input);

    printf("\n");

    switch(input)   /* checking upper and lower cases */
    {
        case 'O':
        case 'o':
                printf("This cylinder contains ammonia.\n\n");
                break;

        case 'B':
        case 'b':
                printf("This cylinder contains carbon monoxide.\n\n");
                break;

        case 'Y':
        case 'y':
                printf("This cylinder contains hydrogen.\n\n");
                break;

        case 'G':
        case 'g':
                printf("This cylinder contains oxygen.\n\n");
                break;

        default:
                printf("Contents unknown.\n\n");

    }

    return 0;
}


/******************************   CHAPTER 4  -  PROJECT 5   *****************************
 *
 *  Earthquake Characterization Program
 *  Programmer:  John Regnier
 */

#include <stdio.h>      /*  printf, scanf definitions */

/* Prototype */
void instruct(void);

int
main(void)
{
      double      n;    /*  number on the Richter scale     */

      instruct();

      /*  Get number (n) on scale   */
      printf("\nEnter the number on the Richter scale>  ");
      scanf("%lf", &n);

      /*  Characterize number (n)   */
      if (n < 5.0)
```

```
            printf("\n[%.1f] -- Little or no damage.\n", n);
      else if (n < 5.5)
            printf("\n[%.1f] -- Some damage.\n", n);
      else if (n < 6.5) {
            printf("\n[%.1f] -- Serious damage: \n", n);
            printf("       walls may crack or fall.\n");
      } else if (n < 7.5) {
            printf("\n[%.1f] -- Disaster:  \n", n);
            printf("       houses and buildings may collapse.\n");
      } else {
            printf("\n[%.1f] -- Catastrophe:  ", n);
            printf("\n       most buildings destroyed.\n");
      }

      printf("\n");

      return(0);

}

/*  Display Instructions to User    */
void
instruct(void)
{
      printf("\n\n");
      printf(" *** National Earthquake Information Center Program ***\n");
      printf("This program will characterize an earthquake for you.\n");
      printf("Just enter the Richter scale number (n).\n");
      printf("-------------------------------------------------->>>>\n");
}


/*****************************   CHAPTER 4  -  PROJECT 7   *****************************
 *
 *  Jim Anderson
 *
 *  This program determines the quadrant or axis on which a point, described
 *    by a set of x-y coordinates, lies.
 */

#include <stdio.h>

int
main(void)
{
   double x_coord,              /* input - the x coordinate */
          y_coord;              /* input - the y coordinate */

/* Get input data */
   printf("Please enter the coordinates, with the x coordinate preceding\n");
   printf("the y coordinate and a space separating the two> ");
   scanf("%lf%lf", &x_coord, &y_coord);
   printf("\n\n\n");

/* Determine where point is located */
   printf("The point (%.2f, %.2f) lies ", x_coord, y_coord);
   if (x_coord == 0.0) {
           if (y_coord != 0.0)
                printf("on the y-axis.");
           else
                printf("at the origin.");
   } else if (y_coord == 0.0) {
           printf("on the x_axis.");
   } else if (x_coord > 0.0) {
           if (y_coord > 0.0)
                printf("in quadrant I.");
           else
                printf("in quadrant IV.");
   } else if (x_coord < 0.0) {
           if (y_coord > 0.0)
                printf("in quadrant II.");
           else
                printf("in quadrant III.");
   }

   printf("\n");
   return (0);
}



/*****************************   CHAPTER 5  -  PROJECT 9   *****************************
 *
 * Programmer: Amy Fletcher
 */

#include <stdio.h>                      /* Definitions of scanf and printf  */
#define LITER 0.001                     /* converting milliliters to liters */

/* This program uses Van der Waals equation of state of a gas to find the
 * pressure of a given amount of Carbon dioxide at a given temperature.
 */

/* Prototype */
void instruct(void);

int
main(void)
{
    double moles,                         /* amount of carbon dioxide         */
           press,                         /* pressure of carbon dioxide       */
           temp,                          /* temperature value for volume     */
           first_vol,                     /* initial volume of CO2            */
           final_vol,                     /* final volume of CO2              */
           step,                          /* increment value for volume       */
           count_vol;                     /* loop control variable            */

    /* Display user instructions                                              */
    instruct();

    /* Get values of inputs                                                   */
    printf("\n\nAmount of Carbon Dioxide(moles)>");
    scanf("%lf", &moles);
    printf("Temperature (Kelvins)>");
    scanf("%lf", &temp);
    printf("Initial volume(milliliters)>");
    scanf("%lf", &first_vol);
    printf("Final volume(milliliters)>");
    scanf("%lf", &final_vol);
    printf("Volume increment(milliliters)>");
    scanf("%lf", &step);

    /* Display initial conditions                                              */
    printf("\n\n%.4f moles of carbon dioxide\n", moles);
    printf("at an absolute temperature of %.2f degrees\n\n", temp);

    /* Display the table heading                                               */
    printf("  Volume          Pressure\n");
    printf("  (milliliters)   (atmospheres)\n");

    /* Display the table                                                       */
    for ( count_vol = first_vol;            /* initialization                */
          count_vol <= final_vol;           /* loop repetition condition     */
          count_vol = count_vol + step) {    /* update                        */
        press = (0.08206 * moles * temp) / (count_vol * LITER - 0.0427 * moles)
              - (3.592 * moles * moles) / (count_vol * LITER * count_vol * LITER);
        printf("    %.2f            %.4f\n", count_vol, press);
    }

    return (0);
}

/*
 * Display user instructions.
 */
void
instruct(void)
{
    printf("\nPlease enter at the prompts the number of moles of carbon ");
    printf("dioxide,\nthe absolute temperature, the initial volume in ");
    printf("milliliters, the\nfinal volume, and the increment volume ");
    printf("between lines of the table.");
}


/*****************************   CHAPTER 5  -  PROJECT 10   *****************************
 *
 *  Russell Updike
 *
 * This program uses an iterative approach to determine how deep the water
 * in a given channel will be when 1000 cubic feet per second is flowing
 * through it.
 */

#include <stdio.h>
#include <math.h>

/* Constant macros */
#define   WIDTH       15.0      /* Width of channel (ft)                         */
#define   DEPTH       10.0      /* Depth of channel (ft)                         */
#define   SLOPE         .0015   /* Slope of channel (ft/ft)                      */
#define   ROUGH_COEF    .014    /* Roughness coefficient                         */
#define   TARGET    1000.0      /* Target flow rate (cubic feet per sec)         */
#define   ACCURACY      .001    /* Allowable error rate                          */

int
main(void)
{
         double water_depth,    /* input - guess at depth of water               */
                q,              /* calculated flow rate (cubic ft/sec)           */
                r,              /* hydraulic radius                              */
                area,           /* cross sectional area of water                 */
                diff;           /* difference between target and computed flow */

       /* Compute and display flow for half the channel depth */
       area = (DEPTH / 2.0) * WIDTH;
       r = area / (2.0 * (DEPTH / 2.0) + WIDTH);
       q = (1.486 / ROUGH_COEF) * area * pow(r, 2.0 / 3.0) * sqrt(SLOPE);
       diff = TARGET - q;
       printf("\nAt a depth of %.1f feet, the flow is %.4f cubic feet per second.\n",
              DEPTH / 2.0, q);
       printf("\nEnter your initial guess (0 < guess < 10.0) for the water depth\n");
       printf("when a flow of %.1f cubic feet per second is desired.\n", TARGET);

       /* Get guesses and compute flow until within .1 percent of target */
       while (fabs(diff) > ACCURACY * TARGET) {
             printf("\nEnter guess> ");
             scanf("%lf", &water_depth);
             area = water_depth * WIDTH;
             r = area / (2.0 * water_depth + WIDTH);
             q = (1.49 / ROUGH_COEF) * area * pow(r, 2.0 / 3.0) * sqrt(SLOPE);
             diff = TARGET - q;
             printf ("\nDepth:  %f ft       Flow:  %.4f cf/s       Target:  %.1f cf/s\n",
                     water_depth, q, TARGET);
             printf("Difference:  %.4f          Error:  %.4f percent \n", diff,
                     diff / 10.0);
       }
       printf("GOOD GUESS!");

       return (0);
}


/******************** CHAPTER 6 - PROJECT 2 ***********************
 *
 ***** Paul Onakoya *****
 *
 This program displays the following numbered menu:



  Enter the number of the problem you wish to solve.

   GIVEN A MEDICAL ORDER IN                  CALCULATE RATE IN

   [1] ml/hr & tubing drop factor            drops/min
   [2] 1 L for n hr                          ml/hr

   [3] mg/kg/hr & concentration in mg/ml     ml/hr

   [4] units/hr & concentration in units/ml  ml/hr

   [5] Quit

 and prompts the user to select an option. If the user enters 1, it prompts
 for the rate in ml/hr and the tubing's drop factor then displays intravenous
 rate in drops per minute. If 2 is selected,number of hours for 1 liter is prompted
 for then Intra. Rate in ml/hr is displayed for this option (2 through 4). If 3 is
 selected, rate in mg/kg/hr, patient's weight in kg, and concentration in mg/ml are
 prompted for, and Intra. Rate is displayed. If 4 is selected, rate in units/hr and
 concentration in units/ml are prompted for, and Intra. Rate is displayed. Option 5
 is to quit the program.*/


#include <stdio.h>

#define MINUTE 60      /* number of minutes in an hour*/
#define M_LITER 1000
#define SENTINEL 5

int get_problem(void);

void get_rate_drop_factor(double *, double *);

void get_kg_rate_conc(double *, double *, double *);

void get_units_conc(double *, double *);

double fig_drops_min(double, double);

double fig_ml_hr(double);

double by_weight(double, double, double);

double by_units(double, double);


int
main(void)
{
   int value;         /* option choosen*/
   double answer;         /* return value of functions*/
   double ml_hour;      /* rate in ml/hr*/
   double drops_ml;      /* tubing factor in drops/ml*/
   double mg_kg_hour;      /* rate in mg/kg/hr*/
   double pat_weight;      /* patient's weight in kg*/
   double mg_ml;         /* concentration in mg/ml*/
   double units_hour;      /* rate in units/hr*/
   double units_ml;      /* concentration in units/ml*/
   double num_hours;      /* number of hours for 1 L to be delivered*/

   value = get_problem();

   while( value != SENTINEL )
   {
      switch(value)
      {
         case 1: get_rate_drop_factor(&ml_hour, &drops_ml);
                 answer = fig_drops_min(ml_hour, drops_ml);
                 printf("\n");
                 printf("The drop rate per minute is %.0f\n\n", answer);
                 break;

         case 2: printf("Enter number of hours => ");
                 scanf("%lf", &num_hours);
                 printf("\n");
                 answer = fig_ml_hr(num_hours);
                 printf("The rate in milliliters per hour is %.0f\n\n", answer);
                 break;

         case 3: get_kg_rate_conc(&mg_kg_hour, &pat_weight, &mg_ml);
```

```
                 answer = by_weight(mg_kg_hour, pat_weight, mg_ml);
                 printf("\n");
                 printf("The rate in milliliters per hour is %.0f\n\n", answer);
                 break;

         case 4: get_units_conc(&units_hour, &units_ml);
                 answer = by_units(units_hour, units_ml);
                 printf("\n");
                 printf("The rate in milliliters per hour is %.0f\n\n", answer);
                 break;

         default: printf("Wrong input.\n");

      }

      value = get_problem();

   }
}

/* function displays menu and gets user's input*/

int
get_problem(void)
{

   int menu_number;
   printf("Enter the number of the problem you wish to solve.\n\n");
   printf("GIVEN A MEDICAL ORDER IN\t\t\tCALCULATE RATE IN\n\n");
   printf("[1] ml/hr & tubing drop factor\t\t\tdrops/min\n");
   printf("[2] 1 L for n hr\t\t\t\tml/hr\n");
   printf("[3] mg/kg/hr & concentration in mg/ml\t\tml/hr\n");
   printf("[4] units/hr & concentration in units/ml\tml/hr\n");
   printf("[5] Quit\n\n");

   printf("Problem => ");
   scanf("%d", &menu_number);

   return menu_number;

}

/* function prompts the user to enter rate and tubing's drop factor then returns
 * values through output parameters
 */
void
get_rate_drop_factor(double *ml_hour, double *drops_ml)
{
   printf("Enter rate in ml/hr => ");
   scanf("%lf", ml_hour);
   printf("\n");
   printf("Enter tubing's drop factor(drops/ml) => ");
   scanf("%lf", drops_ml);
   printf("\n\n");
}


/* function prompts for rate, patient's weight, and concentration then returns
 * values through output parameters
 */
void
get_kg_rate_conc(double *mg_kg_hour, double *pat_weight, double *mg_ml)
{
   printf("Enter rate in mg/kg/hr => ");
   scanf("%lf", mg_kg_hour);
   printf("\n");
   printf("Enter patient weight in kg => ");
   scanf("%lf", pat_weight);
   printf("\n");
   printf("Enter concentration in mg/ml => ");
   scanf("%lf", mg_ml);
   printf("\n\n");
}

/* function prompts for rate and concentration then returns
 * values through output parameters
 */
void
get_units_conc(double *units_hour, double *units_ml)
{
   printf("Enter rate in units/hr => ");
   scanf("%lf", units_hour);
   printf("\n");
   printf("Enter concentration in  units/ml => ");
   scanf("%lf", units_ml);
   printf("\n\n");
}

/* function takes as input rate and concentration then returns as its value the result of
 * dividing their product by MINUTE
 */
double
fig_drops_min(double ml_hour, double drops_ml)
{
   return ml_hour * drops_ml / (double)MINUTE;
}




/* function takes as input num_hours and returns as its value the quotient of 1000
 * and  num_hours
 */
double
fig_ml_hr(double num_hours)
{
   return M_LITER / num_hours;
}

/* function takes 3 inputs and returns as its value the product of rate and patient's
 * weight divided by concentration
 */
double
by_weight(double mg_kg_hr, double pat_weight, double mg_ml)
{
   return mg_kg_hr * pat_weight / mg_ml;
}

/* function takes 2 inputs and returns as its value the quotient of units_hr and units_ml.*/
double
by_units(double units_hour, double units_ml)
{
   return units_hour / units_ml;
}


/*****************************   CHAPTER 6  -  PROJECT 7   ***************************** *
 *
 *  Approximating the square root of a number
 */

#include <stdio.h>
#include <math.h>

#define OK_ERROR 0.005

double square_root (double num);

int
main(void)
{
   printf("The square root of 4 is approximately %.4f\n", square_root(4));
   printf("The square root of 120.5 is approximately %.4f\n",
      square_root(120.5));
   printf("The square root of 88 is approximately %.4f\n", square_root(88));
   printf("The square root of 36.01 is approximately %.4f\n",
      square_root(36.01));
   printf("The square root of 10000 is approximately %.4f\n",
      square_root(10000));
   printf("The square root of 0.25 is approximately %.4f\n", square_root(0.25));
   return (0);
}

/*
 *  Iteratively approximate the square root of num until the difference
 *  between successive approximations is less than OK_ERROR.
 *
 *  Pre: num >= 0
 */
double
square_root (double num)
{
   double next, last = 1.0;

   next = 0.5 * (last + num / last);
   while (fabs(last - next) >= OK_ERROR) {
      last = next;
      next = 0.5 * (last + num / last);
   }

   return (next);
}


/*****************************   CHAPTER 6  -  PROJECT 10  ****************************
 *
 *  Model of a simple calculator
 *     Scans data lines consisting of a binary operator and a right
 *     operand.  Using the current value of the accumulator (initial value
 *     zero) as the left operand, program calculates and displays new
 *     accumulator value.
 *
 *     Recognized operations are addition (+), subtraction (-),
 *     multiplication (*), division (/), and exponentiation (^).
 */

 #include <stdio.h>
 #include <math.h>

 int scan_data (char *operatorp, double *operandp);
 void do_next_op (char op, double operand, double *accump);
 int valid_op (char ch);

 int
 main(void)
 {
    double accum = 0;
    char op;
    int scan_status;
    double operand;

    printf("Repeatedly enter operator and operand. Enter \nq 0\nto quit.\n");
    for (scan_status = scan_data(&op, &operand);
         scan_status == 1;
         scan_status = scan_data(&op, &operand)) {
       do_next_op(op, operand, &accum);
       printf("result so far is %.1f\n", accum);
    }

    printf("final result is %.2f\n", accum);

    return (0);
 }

 /*
  *  Input a one-character operator and a number from a line of data.
  *  If a valid number is scanned and the character is one of these--
  *  + - * / ^ --return 1.  Otherwise return 0.  Discard the rest of the
  *  input line.
  */
 int
 scan_data(char *operatorp, double *operandp)
 {
    char op, next;
    double data;
    int status, return_val;
    status = scanf("%c%lf", &op, &data);
    if (status == 2  &&  valid_op(op)) {
       return_val = 1;
       *operatorp = op;
       *operandp = data;
    } else if (op != 'q' && !valid_op(op)) {
       printf("Invalid operation: %c\n", op);
       return_val = 0;
    } else {
       return_val = 0;
    }

    for (next = op; next != '\n'; scanf("%c", &next)) {}

    return (return_val);
 }

 /*
  *  Verify that ch is one of the characters + - * / ^
  */
 int
 valid_op(char ch)
 {
    int valid;

    switch (ch) {
    case '+':
    case '-':
    case '*':
    case '/':
    case '^':  valid = 1;
               break;

    default:   valid = 0;
    }

    return (valid);
 }

 /*
  *  Apply op using *accum as left operand and operand as right operand,
  *  storing result in *accum.
  *    Pre:  op is + or - or * or / or ^
  */
 void
 do_next_op (char op, double operand, double *accump)
 {
    switch (op) {
    case '+':  *accump = *accump + operand;
               break;

    case '-':  *accump = *accump - operand;
               break;

    case '*':  *accump = *accump * operand;
               break;

    case '/':  *accump = *accump / operand;
               break;

    case '^':  *accump = pow(*accump, operand);
    }
 }


 /*****************************   CHAPTER 7  -  PROJECT 2   *****************************
 *
 *    Demonstration of representational error
 */

 #include <stdio.h>

 #define MAX 30

 int
```

```
 main(void)
 {
    int ct, i;
    double sum, addend;

    for (ct = 2;  ct <= MAX;  ++ct) {
       sum = 0;
       addend = 1.0 / ct;
       for (i = 0;  i < ct;  ++i)
          sum += addend;
       printf("Adding %d  1/%d's gives a result ", ct, ct);
       if (sum == 1.0)
          printf("of 1.\n");
       else if (sum < 1.0)
          printf("less than 1.\n");
       else
          printf("greater than 1.\n");
    }

    return (0);
 }


/*****************************   CHAPTER 8  -  PROJECT 11  *****************************
 *
 *  Joe Fuhrman
 *  binary_search function and driver to test it
 */

#define SIZE_OF_ARRAY 10
#define TRUE 1
#define FALSE 0
#define NOT_FOUND -1

/*       binary_search does a binary search for target in array search_array
 *       over a range of 0 to size - 1.
 *
 *        Pre: search_array, target and size must be defined elsewhere
 */

int binary_srch(const int search_array[],
                int       target,
                int       size)
{
    int top,
        bottom,
        middle,
        index,
        found;

    bottom = 0;
    top = size - 1;
    found = FALSE;

    while ((bottom <= top) && (!found)) {
       middle = bottom + (top - bottom) / 2;
       if (search_array[middle] == target) {
           found = TRUE;
           index = middle;
       } else if (search_array[middle] > target) {
           top = middle - 1;
       } else {
           bottom = middle + 1;
       }
    }

    if (found)
       return (index);
    else
       return (NOT_FOUND);
}



/*****************************   CHAPTER 8  -  PROJECT 15  *****************************
```

```
 *
 *  Joe Fuhrman
 *  an implementation of the Game of Life
 */

#include <stdio.h>

#define ARRAY_SIZE 12    /* size of "universe" array */
#define LIFE 1
#define NO_LIFE 0

void print_array(int life_array[ARRAY_SIZE][ARRAY_SIZE]);
void print_setup_array(int ta[ARRAY_SIZE - 2][ARRAY_SIZE - 2]);
void setup(int life_array[ARRAY_SIZE][ARRAY_SIZE]);
int  count_neighbors(int life_array[ARRAY_SIZE][ARRAY_SIZE], int row, int col);
void swap_arrays(int life_array[ARRAY_SIZE][ARRAY_SIZE], int temp_array[ARRAY_SIZE][ARRAY_SIZE]);
int  birth_survival_death(int life_array[ARRAY_SIZE][ARRAY_SIZE]);
void life(int life_array[ARRAY_SIZE][ARRAY_SIZE]);

int
main(void)
{
    int life_array[ARRAY_SIZE][ARRAY_SIZE];

    setup(life_array);
    life(life_array);

    return (0);
}

/*     print_array is passed a two dimensional integer array to display
 *     to screen.
 *
 *      Pre: life_array must be defined
 */

void
print_array(int life_array[ARRAY_SIZE][ARRAY_SIZE])
{
  int i, j, k;  /* loop control variables */

  for (i = 1;  i < ARRAY_SIZE - 1;  ++i) {
      for (j = 1;  j < ARRAY_SIZE - 1;  ++j) {
          if (life_array[i][j] == LIFE)
              printf(" X ");
          else
              printf("   ");
          if (j != ARRAY_SIZE - 2)
              printf("|");
      }
      if (i != ARRAY_SIZE - 2) {
          printf("\n");
          for (k = 0;  k < ARRAY_SIZE - 2;  ++k) {
              printf("----");
          }
          printf("\n");
      }
  }
  printf("\n");
}

/*      print_setup_array displays the special setup array, allowing user
 *       to see which cells to toggle
 *
 *        Pre: ta must be defined
 */

void
print_setup_array(int ta[ARRAY_SIZE - 2][ARRAY_SIZE - 2])
{
  int i, j, k; /* loop control variables */

  for (i = 0;  i < ARRAY_SIZE - 2;  ++i) {
      for (j = 0;  j < ARRAY_SIZE - 2;  ++j) {
          if (ta[i][j] == -1)
              printf(" XX ");
          else
              printf(" %2d ", ta[i][j]);
          if (j != ARRAY_SIZE - 3)
              printf("|");
      }
      if (i != ARRAY_SIZE - 3) {
          printf("\n");
          for (k = 0;  k < ARRAY_SIZE - 2;  ++k) {
              printf("-----");
          }
          printf("\n");
      }
  }
  printf("\n");
}



/*      setup allows user to setup an initial cell configuration
 *
 *        Pre: life array must be defined
 */
void
setup(int life_array[ARRAY_SIZE][ARRAY_SIZE])
{
    int temp_array[ARRAY_SIZE - 2][ARRAY_SIZE - 2]; /* array to fill */
    int i, j;    /* loop control variables */
    int choice; /* choice of cell to toggle */

    /* fill array with cell numbers */

    for (i = 0;  i < (ARRAY_SIZE - 2);  ++i) {
        for (j = 0;  j < (ARRAY_SIZE - 2);  ++j) {
            temp_array[i][j] = (ARRAY_SIZE - 2)*i + j;
        }
    }

    print_setup_array(temp_array);
    printf("\nToggle -> ");
    scanf("%d", &choice);

    while (choice != -1) {
        if ((choice < -1) || (choice > ((ARRAY_SIZE-2)*(ARRAY_SIZE-2)-1)))
            printf("\nChoice must be between 0 and %d (-1 to stop)\n",
                    ((ARRAY_SIZE - 2)*(ARRAY_SIZE - 2) - 1));
        else {
            i = choice/(ARRAY_SIZE - 2);
            j = choice % (ARRAY_SIZE - 2);
            if (temp_array[i][j] == -1)
                temp_array[i][j] = choice;
            else
                temp_array[i][j] = -1; /* -1 indicates life */
        }
        print_setup_array(temp_array);

        printf("\nToggle -> ");
        scanf("%d", &choice);
    }

    for (i = 0;  i < ARRAY_SIZE;  ++i) {
        for (j = 0;  j < ARRAY_SIZE;  ++j) {
            life_array[i][j] = NO_LIFE;
        }
    }

    for (i = 0;  i < (ARRAY_SIZE - 2);  ++i) {
        for (j = 0;  j < (ARRAY_SIZE - 2);  ++j) {
            if (temp_array[i][j] < 0)
                life_array[i + 1][j + 1] = LIFE;
        }
    }
}

/*
 *      count_neighbors counts the number of neighboring cells
 *      with life in them. It is passed a life array, and the
 *      row and column of the center cell
 *
 *        Pre: life_array, row, col must be defined
 */


int
count_neighbors(int life_array[ARRAY_SIZE][ARRAY_SIZE],
                int row, int col)
{
    int n_count;
    int i, j;

    n_count = 0;

    for (i = row - 1;  i <= row + 1;  ++i) {
        for (j = col - 1;  j <= col + 1;  ++j) {
            if (life_array[i][j] == LIFE)
                n_count += 1;
        }
    }

    if (life_array[row][col] == LIFE)
        n_count -= 1;

    return (n_count);
}

/*
 *      swap_arrays copies temp_array into life_array
 *
 *        Pre: life_array and temp_array must be previously defined
 */
void
swap_arrays(int life_array[ARRAY_SIZE][ARRAY_SIZE],
            int temp_array[ARRAY_SIZE][ARRAY_SIZE])
{
   int i, j;

   for (i = 1;  i < ARRAY_SIZE - 1;  ++i) {
       for (j = 1;  j < ARRAY_SIZE - 1;  ++j) {
           life_array[i][j] = temp_array[i][j];
       }
   }
}

/*
 *      birth_survival_death uses the current generation of life in
 *      life_array to compute which cells in the next generation will
 *      contain life. returns LIFE if at least one cell of the new
 *      generation contains life, otherwise it returns NO_LIFE
 *
 *        Pre: life_array must defined and inititialized
 */
int
birth_survival_death(int life_array[ARRAY_SIZE][ARRAY_SIZE])
{
    int temp_array[ARRAY_SIZE][ARRAY_SIZE], /* next generation life_array */
        i, j,                               /* loop control variables */
        life_indicator = NO_LIFE,           /* return value */
        neighbors;                          /*number of neighbors of a cell*/

    for (i = 1;  i < ARRAY_SIZE -1;  ++i) {
        for (j = 1;  j < ARRAY_SIZE - 1;  ++j) {

            /*initialize temp_array elements */
            temp_array[i][j] = NO_LIFE;

            /* get neighbor count for current cell */
            neighbors = count_neighbors(life_array, i, j);
            /* using the rules of life, determine if a cell is "born",
               survives or dies */

            /* birth */
            if ((life_array[i][j] == NO_LIFE) && (neighbors == 3)) {
                temp_array[i][j] = LIFE;
                life_indicator = LIFE;
            /* survival */
            } else if ((life_array[i][j] == LIFE) &&
                    ((neighbors == 2) || (neighbors == 3))) {
                     temp_array[i][j] = LIFE;
                     life_indicator = LIFE;
            /* death */
            } else if ((life_array[i][j] == LIFE) &&
                    ((neighbors < 2) || (neighbors > 3))) {
                     temp_array[i][j] = NO_LIFE;
            }
        }
    }
    swap_arrays(life_array, temp_array);

    return (life_indicator);
}

/*
 *       life plays the game. displays out successive generations while
 *       life exists.
 *
 *         Pre: life_array must be previously defined and initialized
 */
void
life(int life_array[ARRAY_SIZE][ARRAY_SIZE])
{
    int  life_exists,
         generation = 0;
    char play_on, discard;

    do {
        printf("Life at generation %d\n", generation);
        print_array(life_array);
        ++generation;
        life_exists = birth_survival_death(life_array);

        printf("type 'c' to continue, 'q' to quit\n");
        scanf("%c%c", &discard, &play_on);

    } while ((life_exists == LIFE) && (play_on != 'q'));

    if (life_exists == NO_LIFE)
        printf("Extinction at generation = %d\n", generation);
}


/******************** CHAPTER 9 - PROJECT 4 ***********************
 *
 ***** Paul Onakoya *****
 *
 * This program prompts for the colors of three bands of a resistor, and displays the
 * value of the resistance in kilo-ohms based on a color code
 */

#include <stdio.h>
#include <math.h>    /* for pow*/
#include <ctype.h>   /* for toupper*/
#include <string.h>  /* for strlen*/

#define NOT_FOUND -1   /* constants */
#define SUB_1 10
#define SUB_2 7

int search(const char [][SUB_2], const char [], int);


int
main(void)
{
   char reply,     /* user reply*/
        char_left; /* character left in input stream*/

   do
   {
      int i;               /* counters*/
      int counter;
      int value;            /* subscript of target found in list*/
      double answer = 0.0;   /* value of resistor in kilo-ohms*/
      int no_error = 1;      /* denotes no error*/

      /* initializing the array*/
      char COLOR_CODES[SUB_1][SUB_2] = { "black", "brown", "red", "orange", "yellow",
                                 "green", "blue", "violet", "gray", "white"};

      char target[SUB_2];  /* target string array*/

      printf("Enter the colors of the resistor's three bands, beginning with\n");
      printf("the band nearest the end. Type the colors in lowercase letters only, ");
      printf("NO CAPS.\n\n");

      for(counter = 1; counter < 4 && no_error; counter++)
      {
         printf("Band %d => ", counter);
         scanf("%s", target);
         value = search(COLOR_CODES, target, SUB_1);   /* searches for string*/

         if(value != NOT_FOUND)
         {
            switch(counter)
            {
               case 1:   answer = value * 10;
                         break;

               case 2:   answer += value;
                         break;

               case 3:   if(value > 3)
                            answer *= pow(10, (value - 3));
                         else
                            for(i = 0; i < (3 - value); i++)
                                answer /= 10;
             }
         }

         else
            no_error = 0;      /* if string not found*/
      }

      if(no_error)
         printf("Resistance value: %.3f kilo-ohms\n\n", answer);
      else
         printf("Invalid Color: %s\n\n", target);

      printf("Do you want to decode another resistor?\n => ");
      scanf("%c%c", &char_left, &reply);

      printf("\n");

   } while(toupper(reply) == 'Y');
}



/* function takes as input a list of strings, its size, and a target string.
 * It searches the list for the target and returns as its value the subscript of the
 * target in the list. it returns -1 if target is not found.
 */
int
search(const char COLOR_CODES [][SUB_2], const char target [], int size)
{

   int i,            /* counters*/
       j,
       length,
       counter = 0,
       found = 0,    /* indicates when string is found*/
       where;        /* location of target*/

   length = strlen(target);
   for(i = 0; i < size && !found ; i++)
   {
      for(j = 0; j < length; j++)
         if(COLOR_CODES[i][j] == target[j])
            counter++;

      if(counter == length)
         found = 1;
      else
         counter = 0;
   }

   --i;

   if(found)
      where = i;
   else
      where = NOT_FOUND;

   return where;
}


/*****************************   CHAPTER 9  -  PROJECT 5   *****************************
 *
 *  Takes nouns as input and forms their plurals based on these rules:
 *  a.  If noun ends in "y", remove the "y" and add "ies".
 *  b.  If noun ends in "s", "ch" or "sh", add "es".
 *  c.  Otherwise just add "s".
 */

#include <stdio.h>
#include <string.h>

#define MAX_LEN 20
#define SENT "exit"

char *pluralize(char *plural, const char *noun, int pl_size);

int
main(void)
{
      char noun[MAX_LEN], plural[MAX_LEN];
      int i;

      printf("\nEnter %s to quit.", SENT);
      printf("\nEnter a singular noun to be put into plural form:  ");

      /* Sentinel controlled loop for string input. */
      for  (scanf("%s", noun);
            strcmp(noun,SENT) != 0;
            scanf("%s", noun)) {
          pluralize(plural, noun, MAX_LEN);
          printf("\n%s %s", noun, plural);
          printf("\n\nEnter a singular noun to be put into plural form:  ");
      }
      return (0);
}

char *
pluralize(char       *plural,  /* output string          */
          const char *noun,    /* input string           */
          int         pl_size) /* input - size of plural */
{
      char end[3];
      int last;

      /* Set last equal to the value of the string's length. */
      last = strlen(noun);

      /* Copy the last two characters of noun into end. */
      strcpy(end, &noun[last - 2]);

      /* Add appropriate endings to noun to form plurals. */

      if (noun[last-1] == 'y') {
```

```
      /* If noun ends in 'y', delete 'y' and add 'ies' */
             strncpy(plural, noun, last - 1);
             plural[last - 1] = '\0';
             strcat(plural, "ies");
      } else if ((noun[last-1] == 's') || ((strcmp(end,"ch") == 0) ||
                 (strcmp(end,"sh") == 0))) {
             /* noun ends in 's', 'sh', or 'ch', so add 'es' */
             strcpy(plural, noun);
             strcat(plural, "es");
      } else { /* otherwise add an 's' */
             strcpy(plural, noun);
             strcat(plural, "s");
      }
      /* return the plural form of the noun */
      return (plural);
}


/*****************************   CHAPTER 10  -  PROJECT 2  *****************************
 *
 *  Chris Hansen
 */
#include <stdio.h>
#include <string.h>

#define MAX_LEN 25
#define SENT "exit"

/* Prototypes */
int palindrome(const char *string) ;

int
main(void)
{
      char string[MAX_LEN];

      printf("This program will test to see if a string is a palindrome.\n");
      printf("Please input a deblanked, unpunctuated string of characters.\n");
      printf("Enter %s to stop.\n\nInput:  ", SENT);

      for  (scanf("%s", string);
            strcmp(string, SENT);
            scanf("%s", string)) {
          printf("\nInput:  %s", string);
          if (palindrome(string))
               printf("%s is a palindrome.\n");
          else
               printf("%s is not a palindrome.\n");
          printf("\nInput:  ");
      }
}

/*
 *  Returns a value of 1 if the string argument is a palindrome.
 *  Notice that level, deed, sees, Madam I'm adam (madamimadam)
 *  are palindromes because the first letter matches the last, etc.
 */
int
palindrome(const char *string)
{
      int ans;
      char sub[MAX_LEN];

      if (strlen(string) <= 1)  {
            ans = 1;
      } else if (string[0] != string[strlen(string)-1])  {
            ans = 0;
      } else  {
            strncpy(sub, &string[1], strlen(string) - 2);
            sub[strlen(string) - 2] = '\0';
            ans = palindrome(sub);
      }

      return (ans);
}
```

```
/*****************************   CHAPTER 10  -  PROJECT 5  *****************************
 *
 *  Chris Hansen
 *
 *  Program accepts an N_ROWS X N_COLS array of characters which represent
 *  a maze.  Lists any path from position (0, 1) to (N_ROWS-1, N_COLS-1).
 *  No diagonal moves are allowed.  If no path exists, a message is ouput.
 */

#include <stdio.h>

#define N_ROWS 8
#define N_COLS 8

#define FOUND 0
#define DEAD_END 1

#define PATH '-'
#define FILLED 'X'
#define EMPTY '.'

/* Prototypes */
int find_path(char grid[N_ROWS][N_COLS], int  j, int i);
int start_maze(char grid[N_ROWS][N_COLS], int j, int i) ;

/*
 *  Gets a two dimensional array representing a maze from
 *  the keyboard and prints a path which goes from the start to the end.
 *  If no path exists, a message stating so is output.
 */
int
main(void)
{
      char grid[N_ROWS][N_COLS];     /* the grid                          */
      int x, y, done;

      printf("Please input the %dX%d array which represents a maze.",
             N_ROWS, N_COLS);
      printf("Enter an '%c' for a filled square, and a '%c' for empty.",
             FILLED, EMPTY);


      for  (y = 0; y <= N_ROWS-1; y++) {
          printf("Input row number %d:  ", y);
          scanf("%s", grid[y]);
      }

      /* Echo the maze which was input */
      for  (y = 0; y <= N_ROWS-1; y++) {
          for  (x = 0; x <= N_COLS-1; x++)
              printf("%c", grid[y][x]);
          printf("\n");
      }

      /* Find the path thru the maze. */
      if (grid[0][1]!=EMPTY)
           done = DEAD_END;
      else
           done = start_maze(grid, 0, 1);
      if (done == DEAD_END)
           printf("\nNo path thru maze.\n");

      /* Print the updated version of grid, which shows the path. */
      printf("\n");
      for  (y = 0; y <= N_ROWS-1; y++) {
          for  (x = 0; x <= N_COLS-1; x++)
              printf("%c", grid[y][x]);
          printf("\n");
      }

      return (0);
}

/*
 *  Movements through the maze are generated by this function. Movements always go
 *  towards the end if possible, and then away if there is no other alternative.
 *  Each movement is output and also changed in grid.
 *
 *  Post: Elements of grid are changed in order to
 *        show that they have been traveled through already.
 */
int
find_path( char grid[N_ROWS][N_COLS],  /* input/output - grid containing maze */
           int  j, int i)              /* input - coordinates of point        */
{
      /* Mark the location as having been traveled in*/
      grid[j][i]=PATH;
      printf("( %d, %d)", j, i);

      /* Decide which direction to go */
      if ((j == N_ROWS-1) && (i == N_COLS-1))
            return (FOUND);
      else if ((j+1 <= N_ROWS-1) && (grid[j+1][i] == EMPTY))
            ++j;
      else if ((i+1 <= N_COLS-1) && (grid[j][i+1] == EMPTY))
            ++i;
      else if ((j-1 >= 0) && (grid[j-1][i] == EMPTY))
            --j;
      else if ((i-1 >= 0) && (grid[j][i-1] == EMPTY))
            --i;
      else
            return (DEAD_END);

      return (start_maze(grid, j, i));
}

/*
 *  Starts the search through the maze from point (j, i).
 *  If a dead-end is found, all subsequent paths are followed from the
 *  previous location.  If all paths lead to dead-ends then there is
 *  no path and control is returned to the routine that called start_maze.
 */
int
start_maze(char grid[N_ROWS][N_COLS], int j, int i)
{
      int done;

      done = find_path(grid, j, i);
      if (done == DEAD_END)
           done = find_path(grid, j, i);
      return (done);
}



/****************************   CHAPTER 10  -  PROJECT 7  ****************************
 *
 *  This program computes all different binary number combinations of
 *    a string given as a series of 1's, 0's, and x's, where the x's
 *    represent either a 1 or a 0.
 *
 *  Assumes access to function is_element from Fig. 10.20
 */

#include <stdio.h>
#include <string.h>
#define MAX_STR_LEN 120
#define TRUE   1
#define FALSE  0

/*
 *  This function replaces a single character of a string with another
 *    character (both supplied by the calling function) the first time
 *    it occurs in the string, leaving the rest of the string intact.
 */
char *
replace_1st (char *string,        /* input/output - string to be modified */
             char  old,           /* input - old character to be replaced */
             char  new)           /* input - replacement character        */
{
  int i = 0,                      /* string subscript                     */
```

```
      found = FALSE;              /* loop control variable                */

  /* finds first occurrence of old and replaces it with new */
  while (i < strlen(string) && !found) {
        if (string[i] == old) {
                string[i] = new;
                found = TRUE;
        } else {
                ++i;
        }
  }

  return (string);
}

/*
 *  This function uses recursion to print all binary combinations possible
 *    from a string of 1's, 0's, and x's given to it by the calling function.
 */

void
find_combos (char *string)          /* input - initial string              */
{
  char str1[MAX_STR_LEN],            /* intermediate strings to hold        */
       str2[MAX_STR_LEN];            /*  copies of string to play with      */

  if (!is_element('x', string)) { /* simple case - no x's */
        printf("%s\n", string);
  } else {
        strcpy(str1, string);  /*  combos with 0 for x */
        replace_1st(str1, 'x', '0');
        find_combos(str1);

        strcpy(str2, string);  /*  combos with 1 for x */
        replace_1st(str2, 'x', '1');
        find_combos(str2);
  }
}


/*****************************   CHAPTER 11  -  PROJECT 4   *****************************
 ***** Paul Onakoya *****
 *
 * This program generates explanations of combination-of-positives triplets through data
 * given in a Table of Bacteria Concentration using Most Probable Number (MPN) method.
 */

#include <stdio.h>
#include <string.h>    /* for strcmp*/
#include <ctype.h>     /* for toupper*/

#define MAX_SIZE 18
#define NOT_FOUND -1

typedef struct
{
   char pos_triplets[6];
   int mpn_number;
   int low_limit;
   int up_limit;
}Bac_conc; /* bacterial concentration*/

int loadMpnTable (char [], Bac_conc [], int);
int search(Bac_conc [], int, const char []);

int
main(void)
{
   char reply,    /* user repsonse*/
        char_left;    /* character left in stream*/
   char target[6];   /* combination-of-positives triplet*/
   Bac_conc mpn_table[MAX_SIZE]; /* array of structures*/
   int actual_size;   /* actual size of array*/
   char file_name[13];
```

```
    int location;  /* subscript of searched string*/

    printf("Enter the name(max. 12 characters) of the file whose data should be read => ");
    scanf("%s", &file_name);

    actual_size = loadMpnTable (file_name, mpn_table, MAX_SIZE);

    do
    {
        printf("Enter a combination-of-positives triplet(max. 5 characters) => ");
        scanf("%s", target);

        location = search(mpn_table, actual_size, target);

        if(location != NOT_FOUND)
        {
            printf("\nFor %s, MPN = %d; 95 percent ", target,
                                mpn_table[location].mpn_number);

            printf("of samples contain between %d and %d bacteria / 100 ml.\n\n",
                mpn_table[location].low_limit, mpn_table[location].up_limit);

        }
        else
            printf("\nThis combination is unavailable.\n\n");

        printf("Enter y to try another combination => ");
        scanf("%c%c", &char_left,&reply);

    } while(toupper(reply) == 'Y');

}

/* function takes as parameters the input file name, the array of structures, and its max. size.
 * function opens the file, fills the array, and closes the file. it returns the actual size of
 * the array as the function result. if file contains too much data, the function stores as much
 * data as will fit, displays an error message indicating some data has been ignored, and returns
 * the array's max. size as its actual size.
 */
int
loadMpnTable (char file_name[], Bac_conc mpn_table[], int max_size)
{
    FILE *infile;
    char line[80];
    int counter = 0;
    int status;
    infile = fopen(file_name, "r");
    while(infile == NULL)
    {
        printf("\nCannot open %s for input.\n\n", file_name);
        printf("Re-enter file name => ");
        scanf("%s", file_name);
        infile = fopen(file_name, "r");
    }

    printf("\nFile opened.\n");
    printf("Loading.........\n\n");

    fgets(line, 80, infile);  /* get first 2 lines of file i.e. file headings */
    fgets(line, 80, infile);

    while(counter < max_size &&  status != EOF)
    {
        status = fscanf(infile, "%s", mpn_table[counter].pos_triplets);
        fscanf(infile, "%d", &mpn_table[counter].mpn_number);
        fscanf(infile, "%d", &mpn_table[counter].low_limit);
        fscanf(infile, "%d", &mpn_table[counter].up_limit);

        if(status != EOF)
            counter++;
    }


    if(status != EOF && counter == max_size)
        printf("Too much data in input file. Some data will be truncated.\n");
    printf("Loading Complete.\n\n");
    fclose(infile);  /* close file*/
    printf("File Closed.\n\n");

    return counter;
}

/* function takes as parameters the array of structures, its actual size and a target string.
 * returns the subscript of the structure whose combination-of-positives component matches the
 * target or NOT_FOUND if not found
 */
int
search(Bac_conc mpn_table[], int actual_size, const char target[] )
{
    int found = 0;
    int where;
    int i;
    for(i = 0; i < actual_size && !found; i++)
    {
        if(strcmp(mpn_table[i].pos_triplets, target) == 0)
            found = 1;
    }
    --i;

    if(found)
        where = i;
    else
        where = NOT_FOUND;
    return where;

}

/*****************************   CHAPTER 11  -  PROJECT 8  *****************************
 *
 *  Joe Fuhrman
 */

#include <stdio.h>

#define ITEM_NAME_SIZE 20 /* max size of product name */
#define MAX_ITEMS       5 /* maximum number of products */

/*typedefs*/

typedef struct {
   int day,
       month,
       year;
} date_t;

typedef struct {
   char    meat_type;
   date_t  packaging_date;
   date_t  expiration_date;
} meats_t;

typedef struct {
   char    produce_type;
   date_t  received_date;
} produce_t;

typedef struct {
   date_t expiration_date;
} dairy_t;

typedef struct {
   date_t  expiration_date;
   int     aisle_number;
   char    aisle_side;
} canned_t;

typedef struct {
   char  category;
   int   aisle_number;
   char  aisle_side;
} non_foods_t;

typedef union {
   meats_t      meats;
   produce_t    produce;
   dairy_t      dairy;
   canned_t     canned_goods;
   non_foods_t  non_foods;
} category_t;

typedef struct {
   char        item_name[ITEM_NAME_SIZE];
   int         unit_cost;
   char        product_category;
   category_t  category;
} item_t;

/* Prototypes */
void print_item(item_t item) ;
item_t get_item(void) ;

int
main(void)   /* driver function */
{
   item_t items[MAX_ITEMS];
   int i;

   for (i = 0;  i < MAX_ITEMS;  ++i) {
      items[i] = get_item();
      getchar();
   }

   for (i = 0;  i < MAX_ITEMS;  ++i) {
      print_item(items[i]);
   }

   return (0);
}

/*
 *     print out information for item
 *        Pre: item must be defined and initialized
 */
void
print_item(item_t item)
{
   printf("\n%s %dC", item.item_name, item.unit_cost);

   switch(item.product_category) {
   case 'M' : printf(" %c %d-%d-%d %d-%d-%d", item.category.meats.meat_type,
                     item.category.meats.packaging_date.day,
                     item.category.meats.packaging_date.month,
                     item.category.meats.packaging_date.year,
                     item.category.meats.expiration_date.day,
                     item.category.meats.expiration_date.month,
                     item.category.meats.expiration_date.year);
              break;

   case 'P' : printf(" %c %d-%d-%d", item.category.produce.produce_type,
                     item.category.produce.received_date.day,
                     item.category.produce.received_date.month,
                     item.category.produce.received_date.year);
              break;

   case 'D' : printf(" %d-%d-%d", item.category.dairy.expiration_date.day,
                     item.category.dairy.expiration_date.month,
                     item.category.dairy.expiration_date.year);
              break;

   case 'C' : printf(" %d-%d %d%c",
                     item.category.canned_goods.expiration_date.month,
                     item.category.canned_goods.expiration_date.year,
                     item.category.canned_goods.aisle_number,
                     item.category.canned_goods.aisle_side);
              break;
   case 'N' : printf(" %c %d%c", item.category.non_foods.category,
                     item.category.non_foods.aisle_number,
                     item.category.non_foods.aisle_side);
   }
}

/*
 *      get_item prompts user for item information, returns the item.
 *        Pre: none
 */
item_t
get_item(void)
{
   item_t item;

   printf("\nEnter product category (M,P,D,C,N) -> ");
   item.product_category = getchar();
   switch (item.product_category) {
   case 'M' : printf("\nEnter item name -> ");
              scanf("%s", item.item_name);
              printf("\nEnter cost of item (in cents) -> ");
              scanf("%d", &item.unit_cost);
              printf("\nEnter meat type (R,P,F) -> ");
              getchar();
              item.category.meats.meat_type = getchar();
              printf("\nEnter date of packaging (#-#-#) -> ");
              scanf("%d-%d-%d", &item.category.meats.packaging_date.day,
                    &item.category.meats.packaging_date.month,
                    &item.category.meats.packaging_date.year);
              printf("\nEnter date of expiration (#-#-#) -> ");
              scanf("%d-%d-%d", &item.category.meats.expiration_date.day,
                    &item.category.meats.expiration_date.month,
                    &item.category.meats.expiration_date.year);
              break;

   case 'P' : printf("\nEnter item name -> ");
              scanf("%s", item.item_name);
              printf("\nEnter cost of item (in cents) -> ");
              scanf("%d", &item.unit_cost);

              printf("\nEnter produce type (F,V) -> ");
              getchar();
              item.category.produce.produce_type = getchar();
              printf("\nEnter date received (#-#-#) -> ");
              scanf("%d-%d-%d", &item.category.produce.received_date.day,
                    &item.category.produce.received_date.month,
                    &item.category.produce.received_date.year);
              break;

   case 'D' : printf("\nEnter item name -> ");
              scanf("%s", item.item_name);
              printf("\nEnter cost of item (in cents) -> ");
              scanf("%d", &item.unit_cost);

              printf("\nEnter expiration date (#-#-#) -> ");
              scanf("%d-%d-%d", &item.category.dairy.expiration_date.day,
                    &item.category.dairy.expiration_date.month,
                    &item.category.dairy.expiration_date.year);
              break;

   case 'C' : printf("\nEnter item name -> ");
              scanf("%s", item.item_name);
              printf("\nEnter cost of item (in cents) -> ");
              scanf("%d", &item.unit_cost);

              printf("\nEnter expiration date (month-year) -> ");
              scanf("%d-%d",
                    &item.category.canned_goods.expiration_date.month,
                    &item.category.canned_goods.expiration_date.year);
              printf("\nEnter aisle number -> ");
              scanf("%d", &item.category.canned_goods.aisle_number);
              printf("\nEnter aisle side (A,B) -> ");
              getchar();
              item.category.canned_goods.aisle_side = getchar();
              break;
   case 'N' : printf("\nEnter item name -> ");
              scanf("%s", item.item_name);
              printf("\nEnter cost of item (in cents) -> ");
              scanf("%d", &item.unit_cost);

              printf("\nEnter non-food type (C,P,O) -> ");
              item.category.non_foods.category = getchar();
              printf("\nEnter aisle number -> ");
              scanf("%d", &item.category.non_foods.aisle_number);
              printf("\nEnter aisle side (A,B) -> ");
              getchar();
              item.category.non_foods.aisle_side = getchar();
    }

   return (item);
}



/******************** CHAPTER 12 PROJECT 1 ***********************
 ***** Paul Onakoya *****
 *
 * This program inputs a file of meteorological data and determines the site with the
 * greatest temperature variation and the site with the highest average wind speed for
 * all the days in the file.
 */

#include <stdio.h>
#include <string.h>      /* for strcmp*/
#include <ctype.h>       /* for toupper*/

#define MAX_SIZE 70      /* for 7 days in 10 sites*/

typedef struct
{
   char site_id_num[5];
   int day_of_month;
   int wind_speed;
   int temperature;
}measured_data_t;

int Load_Table(char [], measured_data_t [], int);      /* function prototypes*/

void Get_Temp_Var(measured_data_t [], int, int *, int *);

void Get_Ave_Windspeed(measured_data_t [], int, int *, double *);

int decrement (int);

int increment (int);

int
main(void)
{
   char reply,      /* user response*/
        char_left;   /* character left in stream*/

   do {
      measured_data_t database[MAX_SIZE];
      char file_name[13];
      int actual_size;
      int temp_var;
      int site;
      double ave_wind_speed;

      printf("Enter the name(max. 12 characters) of the file whose data should be read => ");
      scanf("%s", &file_name);

      actual_size = Load_Table(file_name, database, MAX_SIZE);

      if(actual_size != 0)
      {

         Get_Temp_Var(database, actual_size, &site, &temp_var);
         printf("Site %s has the greatest temperature variation of %d degrees celcius.\n\n",
               database[site].site_id_num, temp_var);

         Get_Ave_Windspeed(database, actual_size, &site, &ave_wind_speed);

         printf("Site %s has the highest average wind speed of %.2f knots.\n\n",
                  database[site].site_id_num, ave_wind_speed);
      }

      else
         printf("No data to be processed.\n\n");

      printf("Press y to continue => ");
      scanf("%c%c", &char_left, &reply);

      printf("\n");

   } while(toupper(reply) == 'Y');


}


/* function takes as parameters a file name, a structured array, and its maximum size. It opens
 * the file, fills the array, and closes the file. it also returns the actual size of the array
 * as function value
 */
int
Load_Table(char file_name[], measured_data_t database[], int max_size)

{
   FILE *infile;
   char line[80];
   char test;
   int counter = 0;
   int status;
   infile = fopen(file_name, "r");

   while(infile == NULL)            /* for absent file*/
   {
      printf("\nCannot open %s for input.\n\n", file_name);

      printf("Re-enter file name => ");

      scanf("%s", file_name);

      infile = fopen(file_name, "r");

   }

   printf("\nFile opened.\n");
   printf("Loading.........\n\n");

   test = getc(infile);

   if(test != EOF)
   {
      fgets(line, 80, infile);   /* get first 2 lines of file i.e. file headings */
      fgets(line, 80, infile);

      while(counter < max_size &&  status != EOF)
      {
         status = fscanf(infile, "%s", database[counter].site_id_num);

         fscanf(infile, "%d", &database[counter].day_of_month);
         fscanf(infile, "%d", &database[counter].wind_speed);
         fscanf(infile, "%d", &database[counter].temperature);

         if(status != EOF)
            counter++;
      }



      if(status != EOF && counter == max_size)      /* if file overflows array*/
         printf("Too much data in input file. Some data will be truncated.\n");
      printf("Loading Complete.\n\n");
   }

   else
      printf("No data in file.\n\n");

   fclose(infile);  /* close file*/

   printf("File Closed.\n\n");

   return counter;

}


/* function takes as parameters a structured array, its actual size, and output parameters:
 * site which is subscript of a site in the array, and temp_var which is the value of temperature
 * variation.
 */
void
Get_Temp_Var(measured_data_t database[], int actual_size, int *site, int *temp_var)
{

   int i,
       j,
       lower_ext,
       upper_ext,
       pre_temp_var;
   char storage[5];
   *temp_var = 0;

   strcpy(storage, database[0].site_id_num);

   for(i = 0; i < actual_size ; i = j)
   {
      lower_ext = 0;
      upper_ext = 0;

      for(j = i; (strcmp(database[j].site_id_num, storage) == 0) && j < actual_size; j++)
      {
         if(j == i)
            lower_ext = database[j].temperature;
         else if(j == increment(i))
         {
            if(database[j].temperature > lower_ext)
               upper_ext = database[j].temperature;
            else
            {
               upper_ext = lower_ext;
               lower_ext = database[j].temperature;
            }
         }
         else
         {
            if(database[j].temperature < lower_ext)
               lower_ext = database[j].temperature;
            else if(database[j].temperature > upper_ext)
                  upper_ext = database[j].temperature;
         }


      }

      if(j < actual_size)
         strcpy(storage, database[j].site_id_num);

      pre_temp_var = upper_ext - lower_ext;


      if(pre_temp_var > *temp_var)
      {
         *temp_var = pre_temp_var;
         *site = decrement(j);

      }
   }
}


/* function takes as parameters a structured array, its actual size, and output parameters: site
 * which is subscript of a site in the array, and ave_wind_speed which is the value of average
 * wind speed.
 */
void
Get_Ave_Windspeed(measured_data_t database[], int actual_size, int *site, double *ave_wind_speed)

{
   int i,
      j;
   int   counter = 0;
   int sum = 0;
   double   pre_ave_wind_speed;
   char storage[5];
   *ave_wind_speed = 0;

   strcpy(storage, database[0].site_id_num);

   for(i = 0; i < actual_size ; i = j)
   {
      for(j = i; (strcmp(database[j].site_id_num, storage) == 0) && j < actual_size; j++,
counter++)
      {
         sum += database[j].wind_speed;
      }

      if(j < actual_size)
         strcpy(storage, database[j].site_id_num);

      pre_ave_wind_speed = sum / (double) counter;


      if(pre_ave_wind_speed >  *ave_wind_speed)
      {
         *ave_wind_speed = pre_ave_wind_speed;
         *site = decrement(j);

      }
      counter = 0;
      sum = 0;
   }
}


/* function returns 1 less than the input parameter as its value*/
int
decrement(int var)
{
     return (var – 1);
}


/* function increments the input parameter by 1 and returns the value*/
int
increment(int var)
{
     return (var + 1);
}


/*****************************   CHAPTER 12  -  PROJECT 2  *****************************
 *  Programmer:  Kenneth Bower
 *
 *  This program takes the name of two sorted text files containing
 *  information of the type element_t and merges them into
 *  a sorted binary file.
 */


#include <stdio.h>      /*  include standard library of functions  */

#define MAX_LEN 256     /*  maximum length of string arrays  */
```

```
#define LEN_NAME 25     /*  length of name string array in structure  */
#define LEN_SYMBOL 5    /*  length of symbol string array in structure  */
#define LEN_CLAS 50     /*  length of classification string array in structure  */


/*  Define structure type element_t which contains information
 *  for one element.
 */
typedef struct element_s {
     int  anum;         /*  atomic number of element  */
     char name[LEN_NAME],     /*  name of element */
        symbol[LEN_SYMBOL],   /*  chemical symbol of element */
        clas[LEN_CLAS]; /*  classification of element  */
     double weight;           /*  atomic weight of element  */
     int electrons[7];  /*  array representing # of electrons in each shell  */
} element_t;

/*  function declarations  */
void file_merge(FILE *in1p, FILE *in2p, FILE *outp);
int  scan_element(element_t *einp);
void print_element(element_t eoutp);
void fprint_element(FILE *out_file, element_t eoutp);
int  fscan_element(FILE *fp, element_t *einp);

/*
 *  Tests the functions file_merge and fscan_element by prompting
 *  for two input filenames and an output filename.  Function
 *  file_merge is then called and the files are closed.
 */
int
main(void)
{

    FILE *f_onep,     /*  pointer to input file #1  */
         *f_twop,           /*  pointer to input file #2  */
         *f_outp;           /*  pointer to output file  */

    char file_one[MAX_LEN],   /*  name of input file #1  */
         file_two[MAX_LEN],   /*  name of input file #2  */
         file_out[MAX_LEN];   /*  name of output file  */

    /*  get name of input file #1 from keyboard  */
    printf("\nEnter the name of the first file >> ");
    scanf("%s",file_one);
    for(f_onep = fopen(file_one,"r");
      f_onep == NULL;
      f_onep = fopen(file_one,"r")) {
         printf("\nSorry!  File not found.  Try again >> ");
         scanf("%s",file_one);
      }

    /*  get name of input file #2 from keyboard  */
    printf("\nEnter the name of the second file >> ");
    scanf("%s",file_two);
    for(f_twop = fopen(file_two,"r");
      f_twop == NULL;
      f_twop = fopen(file_two,"r")) {
         printf("\nSorry!  File not found.  Try again >> ");
         scanf("%s",file_two);
      }

    /*  get name of output file from keyboard  */
    printf("\nEnter the name of the output file >> ");
    scanf("%s",file_out);
    for(f_outp = fopen(file_out,"wb");
      f_outp == NULL;
      f_outp = fopen(file_out,"r")) {
         printf("\nSorry!  Invalid file name.  Try again >> ");
         scanf("%s",file_out);
      }


    /*  merge the two input files  */
    file_merge(f_onep, f_twop, f_outp);

    /*  close the open files  */
    fclose(f_onep);
    fclose(f_twop);
    fclose(f_outp);

    return(0);
}

void
file_merge(FILE *in1p,    /*  pointer to input file #1  */
           FILE *in2p,    /*  pointer to input file #2  */
           FILE *outp)    /*  pointer to output file  */
{
    element_t from_one,   /*  record from input file #1  */
              from_two;   /*  record from input file #2  */

    int  flag_one,        /*  status flag for scans of file #1  */
         flag_two;        /*  status flag for scans of file #2  */

    /*  get first record from each file  */
    flag_one = fscan_element(in1p, &from_one);
    flag_two = fscan_element(in2p, &from_two);

    /*  continue until an end of file or an error is encountered  */
    while(flag_one == 1 && flag_two == 1) {
       /*  put lowest record and get next record */
       if(from_one.anum < from_two.anum) {
        fwrite(&from_one, sizeof (element_t), 1, outp);
        flag_one = fscan_element(in1p, &from_one);
          }
       else if (from_one.anum > from_two.anum) {
        fwrite(&from_two, sizeof (element_t), 1, outp);
        flag_two = fscan_element(in2p, &from_two);
          }

       /*  if same put record and get new records  */
       else {
        fwrite(&from_one, sizeof (element_t), 1, outp);
          flag_one = fscan_element(in1p, &from_one);
        flag_two = fscan_element(in2p, &from_two);
       }

    }

    /*  if no errors occurred, finish merging document  */
    if (flag_one !=0 && flag_two != 0) {
       while(flag_one == 1) {
          fwrite(&from_one, sizeof (element_t), 1, outp);
          flag_one = fscan_element(in1p, &from_one);
          }

       while(flag_two == 1) {
          fwrite(&from_two, sizeof (element_t), 1, outp);
          flag_two = fscan_element(in2p, &from_two);
        }
       }

    /*  Print error message if error occurred in first data file  */
    if (flag_one == 0) {
      printf("\nAn error was encountered in first input file.");
      printf("\nMerge Aborted!!\n");
      }

    /*  Print error message if error occurred in second data file  */
    if (flag_two == 0) {
      printf("\nAn error was encountered in second input file.");
      printf("\nMerge Aborted!!\n");
      }
}

/*
 *  function scan_element scans a value into each of the components
 *  of a structure type element_t.  It returns a value of 1 if each
 *  of the components was filled.  It returns a zero if all of the
 *  components were not filled.  It also returns a value of EOF if
 *  the EOF character was encountered.
 *
 *  Pre:  a pointer to a structure of type element_t must be passed
 *        to scan_element.
 */
int
scan_element(element_t *einp)
{
   int s_flag = 0;      /*  variable for return value  */

   /*  scan each component of the structure  */
   s_flag = scanf("%d %s %s %s %lf %d %d %d %d %d %d %d", &(*einp).anum,
                  (*einp).name,
                  (*einp).symbol,
                  (*einp).clas,
                  &(*einp).weight,
                  &(*einp).electrons[0],
                  &(*einp).electrons[1],
                  &(*einp).electrons[2],
                  &(*einp).electrons[3],
                  &(*einp).electrons[4],
                  &(*einp).electrons[5],
                  &(*einp).electrons[6]);

   /*  set flag to 1 if all components were filled  */
   if (s_flag == 12)
      s_flag = 1;

   /*  set flag to 0 if all component values were not filled  */
   else if (s_flag != EOF)
      s_flag = 0;

   /*  return status flag of function  */
   return (s_flag);
}


/*
 *  function print_element prints the contents of a structure of
 *  type element_t.  Each component is printed using a specific
 *  field width to space the values to allow the calling function
 *  to display a heading to match the output.
 *
 *  Pre:  a pointer to structure type element_t must be passed
 */
void
print_element(element_t eoutp)
{

    int i;  /*  counting variable  */

    /*  print first 5 components of structure  */
    printf("\n%-6d %-10s %-8s %-15s %10.6f   ", eoutp.anum, eoutp.name,
                                                eoutp.symbol, eoutp.clas,
                                                eoutp.weight);

    /*  print last seven components of structure  */
    for(i = 0; i < 7; i++)
        printf("%2d ",eoutp.electrons[i]);

}

/*
 *  function fprint_element prints an element to the given output text
 *  file.
 */
void
fprint_element(FILE * out_file,   /*  input -- file to write to  */
               element_t eoutp)   /*  input -- pointer to structure to write */
{

    int i;  /*  counting variable  */

    /*  print first 5 components of structure  */
    fprintf(out_file, " %d %s %s %s %f ", eoutp.anum, eoutp.name,
                                 eoutp.symbol, eoutp.clas,
                                 eoutp.weight);
    /*  print last seven components of structure  */
    for(i = 0; i < 7; i++)
            fprintf(out_file, "%d ",eoutp.electrons[i]);
      fprintf(out_file,"\n");

}

/*
 *  function fscan_element scans a value into each of the components
 *  of a structure type element_t from an input file.  It returns a
 *  value of 1 if each of the components was filled.  It returns a
 *  zero if all of the components were not filled.  It also returns
 *  a value of EOF if the EOF character was encountered.
 *
 *  Pre:  a pointer to an already opened text file and a pointer to
 *        a structure of type element_t must be passed to fscan_element.
 */
int
fscan_element(FILE *fp, element_t *einp)
{
   int s_flag = 0;      /*  variable for return value  */

   /*  scan each component of the structure  */
   s_flag = fscanf(fp,"%d%s%s%s%lf%d%d%d%d%d%d%d",  &(*einp).anum, (*einp).name,
                   (*einp).symbol, (*einp).clas, &(*einp).weight, &(*einp).electrons[0],
                   &(*einp).electrons[1], &(*einp).electrons[2], &(*einp).electrons[3],
                   &(*einp).electrons[4], &(*einp).electrons[5], &(*einp).electrons[6]);

   /*  set flag to 1 if all components were filled  */
   if (s_flag == 12)
      s_flag = 1;

   /*  set flag to 0 if all component values were not filled  */
   else if (s_flag != EOF)
      s_flag = 0;

   /*  return status flag of function  */
   return (s_flag);
}


/*****************************   CHAPTER 13  -  PROJECT 3  *****************************
 *  Joan C. Horvath
 */

/*****************************   HEADER FILE    ***************************************/

/*
 * This header file defines all the data types and functions
 * needed for the aircraft mechanic scheduling problem in Ch 13 #3.
 *
 */

#include <stdio.h>
#define MAX_CREWS  3      /* Maximum number of available crews         */
#define  MAX_JOBS  25     /* Maximum number of jobs to be scheduled    */

typedef struct {
      int level;         /* Level of maintenance this crew can perform */
      int cost;          /* Dollars per hour this crew costs           */
      int hours_so_far;  /* How many hours crew is scheduled to work   */
} crew_t;

typedef struct {
      int id;            /* ID of aircraft needing maintenance         */
      int level;         /* Level of maintenance required              */
      int hours;         /* Number of hours required to do this job    */
      int crew;          /* Index of crew to do the work               */
      int start_hours;   /* Time at which work will be started         */
} work_t;

/*
 *  Declare function prototypes. Function variables commented in the
 *  actual function code calls.
 */
```

```
extern void
scan_crew_data(crew_t crew[], int *num_crews);

extern void
scan_maintenance_data(work_t maintenance[], int *num_jobs);

extern void
match_crew_to_maintenance(crew_t crew[],  work_t *maintenance, int n_crews, int *min_cost_crew);

extern void
earliest_time(crew_t crew[], work_t *maintenance, int n_crews, int *min_time_crew);


extern void
print_schedule(crew_t crew[], work_t maintenance[], int n_crews, int n_jobs);


/*****************************   IMPLEMENTATION FILE   *******************************/

/*
 * The functions in this file manipulate data to support the scheduling
 * problem performed by the main program.
 *
 */
#include "PP13_3.h"

/*
 *  Following function scans in crew data and returns it to the
 *  calling program.
 */
void
scan_crew_data(
      crew_t crew[],  /* Main structure in which to keep crew data */
      int *num_crews  /* Number of crews read in                   */
      )
{
   int i;

   /* First find out how many crews you have. */
   printf("Enter the number of crews you have. INPUT> ");
   scanf("%d", num_crews);

   while (( 0 >= *num_crews) || (*num_crews > MAX_CREWS) ) {
       printf("Number out of range. Max=%d. Try again. INPUT> ", MAX_CREWS);
       scanf("%d", num_crews);
   }

   /* Now scan the data and initialize hours the crew has worked */
   printf("For each crew, input a skill level and cost per hour.\n");
   for (i = 0;  i < *num_crews;  ++i) {
         scanf("%d%d", &crew[i].level,  &crew[i].cost);
         crew[i].hours_so_far = 0;
    }

}

/*
 *  Following function scans in  data regarding needed maintenance
 *  and returns it to the calling function.
 */
void
scan_maintenance_data(
      work_t maintenance[], /* Main structure in which to keep
                               track of maintenance to be performed    */
      int *num_jobs         /* Number of jobs scanned                  */
                    )
{
    int i;

   /* First find out how many jobs you have. */
   printf("Enter the number of jobs you have. INPUT> ");
   scanf("%d", num_jobs);

   while ( (0  >= *num_jobs) || (*num_jobs > MAX_JOBS)) {
       printf("Number out of range. Max=%d. Try again. INPUT> ", MAX_JOBS);
       scanf("%d", num_jobs);
   }

   /* Now scan the data. */
   printf("For each job, input an id (4-digit integer),  a skill level");
   printf("\nand estimated hours.\n");

   for (i = 0;  i < (*num_jobs);  i++) {
         scanf("%d%d%d", &maintenance[i].id,  &maintenance[i].level,
                 &maintenance[i].hours);
         maintenance[i].crew = 0;
         maintenance[i].start_hours = 0;
    }
}

/*
 *  Following function compares crew qualifications and one
 *  maintenance job to return the index of the lowest-cost crew
 *  that can do the job.
 */
void
match_crew_to_maintenance(
    crew_t crew[],            /* Input: Structure with all crew data      */
    work_t *maintenance,      /* Input: One maintenance job record        */
    int n_crews,              /* Input: Number of crews                   */
    int *min_cost_crew)       /* Output: Lowest cost crew that can do job */
{
    int i;
    int flag;                 /* Test for out of bounds data              */

    flag = 0;
    *min_cost_crew = 0;

    /* Look for the lowest-cost adequately skilled crew to do the job. */
    for (i = 0;  i < n_crews;  ++i) {
        if (  (maintenance->level <= crew[i].level) &&
              ( (crew[i].cost <= crew[*min_cost_crew].cost) ||
                (flag == 0) )  /* take a more expensive crew if
                                  none yet selected */
           ) {
            *min_cost_crew = i;
            flag = 1;
        }
    }
    if (flag == 0) {
        printf("No crew of adequate level available. Program exiting\n");
        exit (1);
    }

}

/*
 *  Following function compares crew qualifications and one
 *  maintenance job to return the index of the first available crew
 *  that can do the job.
 */
void
earliest_time(
    crew_t crew[],            /* Input: Structure with all crew data      */
    work_t *maintenance,       /* Input: One maintenance job record        */
    int    n_crews,           /* Input: Number of crews                   */
    int   *min_time_crew)     /* Output: First avialable crew that
                                         can do job */
{
    int i;
    int flag;                  /* Tests for out-of-bounds data            */
    flag = 0;
    *min_time_crew = 0;

    /* Look for the next-available adequately skilled crew to do the job. */
    for (i = 0;  i < n_crews;  ++i) {
        if (
              (crew[i].level >= maintenance->level)
               &&
               ((crew[i].hours_so_far < crew[*min_time_crew].hours_so_far)
                 || (flag == 0)
                )
           ) {
            *min_time_crew = i;
            flag = 1;
        }

        /* Check to see if there are two crews who could start  job at the
           same time but one might be cheaper.
         */
        else if (
                 (crew[i].hours_so_far == crew[*min_time_crew].hours_so_far)
              && (crew[i].level >= maintenance->level)
              &&  ((crew[i].cost < crew[*min_time_crew].cost)|| (flag == 0) )
                 ) {
               *min_time_crew = i;
               flag = 1;
         }
      }
       if (flag == 0) {
        printf("No crew of adequate level available. Program exiting\n");
        exit (1);
    }
}

void
print_schedule(
     crew_t crew[],          /* Input: job structure                     */
     work_t maintenance[],   /* Input: crew structure                    */
     int    n_crews,         /* Input: number of crews                   */
     int    n_jobs)          /* Input: number of jobs                    */
{
      int i;
      int latest_finish;     /* Time at which the last job ends          */
      int end_time;          /* Time at which each job ends              */
      int total_cost;        /* Cost of all jobs                         */

      printf("CREW SCHEDULES:\nJOB \tSTART TIME\t END_TIME\tCREW\n\n");

      latest_finish = -1;
      total_cost = 0;

      for (i = 0;  i < n_jobs;  ++i) {
          end_time = maintenance[i].start_hours + maintenance[i].hours;
          printf("%4d\t%10d\t%9d\t%4d\n", maintenance[i].id,
                 maintenance[i].start_hours,
                 end_time,
                 maintenance[i].crew);
          total_cost += crew[maintenance[i].crew].cost *
                        maintenance[i].hours;
          if (end_time > latest_finish)
              latest_finish = end_time;

      }
      printf("\n\nALL JOBS FINISHED AT %d AT A COST OF %d\n",
              latest_finish, total_cost);
}

/*****************************   MAIN FUNCTION   **************************************/

/*
 *  This program calls a variety of functions to schedule aircraft
 *  maintenance given a variety of crews who can handle this maintenance.
 *
 *  The program reports two schedules: one which gets the maintenance
 *  done at the lowest possible cost, the other which gets it done
 *  as cheaply as possible.
 */
#include "PP13_3.h"

int
main(void)
{
      crew_t crew[MAX_CREWS];      /* Main crew data structure        */
      work_t jobs[MAX_JOBS];       /* Main work data structure        */
      int    n_crews;              /* Actual numbers of crews         */
      int    n_jobs;               /* Actual number of jobs           */
      int    output_crew;          /* Crew for a given job            */
      int    order[MAX_JOBS];      /* Array for sorted indices of
                                      the maintenance structure       */
      int    temp;                 /* Temporary sort variable         */
      char   testchar[10];         /* Variable to hold pausing string */
      int    i,j;

      /* Call functions to read in the data.                          */
      read_crew_data(crew, &n_crews);
      read_maintenance_data(jobs, &n_jobs);

      /* Now compute the schedule for the lowest_cost solution. */
      for (i = 0;  i < n_jobs;  ++i) {
          match_crew_to_maintenance(crew, &jobs[i], n_crews, &output_crew);
          jobs[i].crew = output_crew;
          jobs[i].start_hours = crew[output_crew].hours_so_far;
          crew[output_crew].hours_so_far += jobs[i].hours;
      }

      printf("\n\nLOWEST COST SCHEDULE:\n\n");
      print_schedule(crew, jobs, n_crews, n_jobs);


      /* Reinitialize scheduling parameters in the two data structs.*/
      for (i = 0;  i < n_jobs;  ++i) {
          jobs[i].crew = -1;
          jobs[i].start_hours = 0;
      }
      for (i = 0; i < n_crews;  ++i)
          crew[i].hours_so_far = 0;

     /* Ask the user to hit any character to pause the
      * program so that one schedule can be viewed on the
      * screen before it scrolls off.
      */
     printf("Input any character to see the next schedule.\n");
     scanf("%s", &testchar);

     /* Now compute the schedule for the fastest solution.
     /* First sort the jobs into highest-cost first since
      * we want to make the selection process single-pass and
      * to do the sort on job time we want to schedule the
      * most constrained jobs first.
      * Assume the highest-cost jobs are the highest skill levels.
      */
      for (i = 0; i< n_jobs; ++i)
          order[i] = i;

      for (i = 0; i < n_jobs; ++i) {
          for (j = i; j < n_jobs; ++j) {
              if (jobs[i].level < jobs[j].level) {
              temp =order[i];
              order[i] = order[j];
              order[j] = temp;
              }
          }
      }

      for (i = 0;  i < n_jobs;  ++i) {
            j = order[i];

            earliest_time(crew, &jobs[j], n_crews, &output_crew);
            jobs[j].crew = output_crew;
            jobs[j].start_hours = crew[output_crew].hours_so_far;
            crew[output_crew].hours_so_far += jobs[j].hours;
      }

      printf("\n\nFASTEST SCHEDULE:\n\n");
      print_schedule(crew, jobs, n_crews, n_jobs);

      return (0);
}


/****************************   CHAPTER 14  -  PROJECT 4  ****************************
```

```
 *  Mark A. Thoney
 */

/*****************************   HEADER FILE    ***************************************/
/*
 * stack.h
 *
 * This file contains the data type declaration and
 * function declarations for maintaining a stack.
 *
 */

#include <stdio.h>
#include <stdlib.h>

typedef int stack_element_t;

typedef struct stack_node_s {       /* stack declaration */
      stack_element_t   element;
      struct stack_node_s *restp;
} stack_node_t;

typedef struct {
      stack_node_t *topp;
} stack_t;

/* Headers */
extern void push(stack_t *sp, stack_element_t operand);
extern stack_element_t pop(stack_t *sp);
extern stack_element_t retrieve(stack_t sp);


/*****************************   IMPLEMENTATION FILE   ********************************/
/*
 * stack.c
 *
 *  Creates and manipulates a stack of stack elements
 */

#include "stack.h"

/*
 *  The value in c is placed on top of the stack accessed through sp
 *  Pre:  the stack is defined
 */
void
push(stack_t *sp,              /* input/output - stack          */
     stack_element_t operand)  /* input        - element to add */
{
      stack_node_t *newp;  /* pointer to new stack node */

      /*  Creates and defines new node                  */
      newp = (stack_node_t *)malloc(sizeof (stack_node_t));
      newp->element = operand;
      newp->restp = (*sp).topp;

      /*  Sets stack pointer to point to new node       */
      (*sp).topp = newp;
}

/*
 *  Removes and frees top node of stack, returning character value
 *  stored there.
 *  Pre:  the stack is not empty
 */
stack_element_t
pop(stack_t *sp) /* input/output - stack */
{
      stack_node_t    *to_freep;  /* pointer to node removed */
      stack_element_t  ans;       /* value at top of stack   */

      to_freep = (*sp).topp;         /* saves pointer to node being deleted */
      ans = to_freep->element;       /* retrieves value to return           */
      (*sp).topp = to_freep->restp;  /* deletes top node                    */
      free(to_freep);                /* deallocates space                   */
      return (ans);
}

/*
 * The value at the top of the stack is returned as the
 * function value.  The stack is not changed.
 * Pre: s is not empty
 */
stack_element_t
retrieve(stack_t sp) /* input */
{
  return(sp.topp->element);
}

/*
 * Test for empty stack.  Return 1 if stack is not empty and
 * 0 if it is
 */
int
is_empty_stack(stack_t sp) /* input */
{
  if (sp.topp == NULL)
    return(0);
  else
    return(1);
}

/*****************************   MAIN FUNCTION   **************************************/
 *
 *  Mark Thoney
 *
 *  This program scans an postfix expression and evaluates it.  It uses
 *  the stack library.  The Postfix expression must be followed by
 *  a space and a '?' to to mark the end of the expression.
 *
 *        i.e.  5 6 * ?
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "stack.h"

int
main(void)
{
  char postexp[30]; /* the variable that will stored the postfix expression
                       that was scanned */
  stack_t sp;       /* the stack */
  stack_element_t l_operand,     /* store operands popped from the stack in */
                  r_operand,     /* these variables                         */
                  operand;       /* used to store operands found in postexp */

  sp.topp = NULL;   /* initialize the stack */
  scanf("%s",postexp);
  while (postexp[0] != '?')
  {
    if (isdigit(postexp[0]))
    {
      sscanf(postexp,"%d",&operand);
      push(&sp, operand);
    } else {
      r_operand = pop(&sp);
      l_operand = pop(&sp);

      switch (postexp[0]) {
      case '+' : push(&sp, (l_operand + r_operand));
             break;

      case '-' : push(&sp, (l_operand - r_operand));
             break;

      case '*' : push(&sp, (l_operand * r_operand));
             break;

      case '/' : push(&sp, (l_operand / r_operand));
```

```
             break;

      }
    }
    scanf("%s",&postexp);
  }

  printf("The result = %d\n",retrieve(sp));

  result (0);
}

/*****************************   CHAPTER 15  -  PROJECT 1  *****************************/
/*                                                                                      */
/* Write a program that creates a child process.  In the child process, sleep for 5   */
/* seconds and then display the message "Child Finished".  In the parent process,     */
/* wait for the child process to finish, then display the message "Parent Finished".  */
/* Call sleep with an argument of 5 in the child process to sleep for 5 seconds       */
/*                                                                                      */
/****************************************************************************************/

#include <stdio.h>
#include <unistd.h>
#include <wait.h>

int
main(void)
{
    printf("Parent Started\n");

    if (fork() == 0)
    {
        printf(" Child Started\n");
        sleep(5);
        printf(" Child Finished\n");
    }
    else
    {
        wait(NULL);
        printf("Parent Finished\n");
    }

    return 0;
}

/*****************************   CHAPTER 15  -  PROJECT 2  *****************************/
/*                                                                                      */
/* Write a program that creates a pipe and then a child process.  In the parent       */
/* process write the character string "Hello World" to the pipe and wait for the      */
/* child process to finish.  In the child process, read the character string from the */
/* pipe and display the character string                                               */
/*                                                                                      */
/****************************************************************************************/

#include <stdio.h>
#include <unistd.h>
#include <wait.h>

int
main(void)
{
    int filedes[2];
    char value[12];

    pipe(filedes);

    if (fork() == 0)
    {
        close(filedes[1]);
         read(filedes[0], value, 12);

        printf("%s\n", value);
    }
    else
    {
        close(filedes[0]);
        write(filedes[1], "Hello World", 11);

        wait(NULL);
    }

    return 0;
}

/****************************   CHAPTER 15  -  PROJECT 3  *****************************/
/*                                                                                    */
/* Write a program that sleeps for 5 seconds and then displays the message "Child     */
/* Finished".  Write a separate program that launches the first program, waits for    */
/* the child process to finish then displays the message "Parent Finished".           */
/*                                                                                    */
/**************************************************************************************/

/* Parent Program */

#include <stdio.h>
#include <unistd.h>
#include <wait.h>

int
main(void)
{
    printf("Parent Started\n");

    if (fork() == 0)
    {
        execl("child.exe", "child.exe", NULL);
    }
    else
    {
        wait(NULL);
        printf("Parent Finished\n");
    }

    return 0;
}

/* Child Program */

#include <stdio.h>
#include <unistd.h>

int
main(void)
{
    printf(" Child Started\n");
    sleep(5);
    printf(" Child Finished\n");

    return 0;
}

/****************************   CHAPTER 15  -  PROJECT 4  *****************************/
/*                                                                                    */
/* Write a program that reads a newline-delimited character string from standard      */
/* input and displays the character string.  Write a separate program that creates a  */
/* pipe, assigns the read file descriptor of the pipe to standard input, launches the */
/* first program, writes the character string "Hello World" to the pipe and waits for */
/* the child process to finish.                                                       */
/*                                                                                    */
/**************************************************************************************/

/* Parent Program */

#include <unistd.h>
#include <wait.h>

int
main(void)
{
    int filedes[2];

    pipe(filedes);
    if (fork() == 0)
    {
        dup2(filedes[0], STDIN_FILENO);

        close(filedes[1]);
        close(filedes[0]);

        execl("child.exe", "child.exe", NULL);
    }
    else
    {
        close(filedes[0]);
        write(filedes[1], "Hello World\n", 12);

        wait(NULL);
    }

    return 0;
}

/* Child Program */

#include <stdio.h>

int
main(void)
{
    char text[12];

    fgets(text, 12, stdin);
    printf("%s\n", text);

    return 0;
}

/****************************   CHAPTER 15  -  PROJECT 5  ****************************/
/*                                                                                   */
/* Write a program that creates a new thread.  In the new thread display the message  */
/* "New Thread Started", sleep for 5 seconds then display the message “New Thread     */
/* Finished”.  In the main thread, wait for the new thread to finish then display the */
/* message "Main Thread Finished".                                                    */
/*                                                                                   */
/*************************************************************************************/

#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

void *thread(void *);

int
main(void)
{
    pthread_t tid;

    printf("Main Thread Started\n");

    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, NULL);

    printf("Main Thread Finished\n");

    return 0;
}

void *
thread(void *argument)
{
    printf(" New Thread Started\n");
    sleep(5);
    printf(" New Thread Finished\n");

    return NULL;
}
```

```
/****************************   CHAPTER 15  -  PROJECT 6  ****************************/
/*                                                                                   */
/* Write a program with a global integer variable and a mutex lock that initializes   */
/* the mutex lock and then creates a new thread.  In the new thread lock the mutex,   */
/* increment the global integer variable, display the message "New Thread Data = "    */
/* with the updated value of the global integer variable and unlock the mutex.  In    */
/* the main thread lock the mutex, increment the global integer variable, display the */
/* message "Main Thread Data = " with the updated value of the global integer         */
/* variable, unlock the mutex and wait for the new thread to finish.                  */
/*                                                                                   */
/*************************************************************************************/

#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

void *thread(void *);

int data;
pthread_mutex_t mutex;

int
main(void)
{
    pthread_t tid;

    data = 0;

    pthread_mutex_init(&mutex, NULL);
    pthread_create(&tid, NULL, thread, NULL);

    pthread_mutex_lock(&mutex);
    printf("Main Thread Data = %d\n", ++data);
    pthread_mutex_unlock(&mutex);

    pthread_join(tid, NULL);

    return 0;
}

void *
thread(void *argument)
{
    pthread_mutex_lock(&mutex);
    printf(" New Thread Data = %d\n", ++data);
    pthread_mutex_unlock(&mutex);

    return NULL;
}

/****************************   CHAPTER 15  -  PROJECT 7  ****************************/
/*                                                                                   */
/* Write a program with a global integer flag and a mutex lock.  The program          */
/* initializes the mutex lock and then creates a new thread.  In the new thread, loop */
/* while the global integer flag is 1 incrementing a local counter once per second    */
/* and displaying the message "Count = " with the updated value of the local counter  */
/* from within the loop.  In the main thread wait for the user to enter any key, then */
/* lock the mutex, set the global integer flag to 0, unlock the mutex and wait for    */
/* the new thread to finish.  Hint: Remember that you will need to lock the mutex,    */
/* store the global integer flag in a local integer variable and unlock the mutex     */
/* from within the loop in order to test the loop condition since you cannot lock the */
/* mutex outside of the loop!                                                         */
/*                                                                                   */
/*************************************************************************************/

#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

void *thread(void *);

int flag;
pthread_mutex_t mutex;

int
main(void)
```

```
{
    pthread_t tid;
    char key;

    flag = 1;

    pthread_mutex_init(&mutex, NULL);
    pthread_create(&tid, NULL, thread, NULL);

    scanf(" %c", &key);

    pthread_mutex_lock(&mutex);
    flag = 0;
    pthread_mutex_unlock(&mutex);

    pthread_join(tid, NULL);

    return 0;
}

void *
thread(void *argument)
{
    int count;
    int local;

    count = 0;

    do
    {
        printf("Count = %d\n", ++count);
        sleep(1);

        pthread_mutex_lock(&mutex);
        local = flag;
        pthread_mutex_unlock(&mutex);
    }
    while (local);

    return NULL;
}

/****************************   CHAPTER 15  -  PROJECT 8  *****************************/
/*                                                                                    */
/* Write a program that creates a new thread and passes an integer value into the new */
/* thread as the argument.  In the new thread display the message "New Thread         */
/* Argument = " with the integer argument value.  In the main thread, wait for the    */
/* new thread to finish.                                                              */
/*                                                                                    */
/**************************************************************************************/

#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

void *thread(void *);

int
main(void)
{
    pthread_t tid;
    int arg;

    arg = 10;

    pthread_create(&tid, NULL, thread, &arg);
    pthread_join(tid, NULL);

    return 0;
}

void *
thread(void *argument)
{
    printf("New Thread Argument = %d\n", *(int*) argument);
    return NULL;
```

```
}



//****************************   CHAPTER 16  -  PROJECT 3  *****************************
//
//  Implement and test a class to represent can objects
//
//  header file can.h
//
#ifndef CAN_H
#define CAN_H

#include <iostream>
using namespace std;

class Can   {

public:
   Can ()  {}        // Default constructor
   Can ( double, double, double ); // Constructor  2 -
                                   //   components initialized
   int capacity( double volume ) const;

private:
   double empty_weight;   // grams
   double base_radius, height;  // centimeters

friend istream& operator>> (istream& is, Can& acan);
friend ostream& operator<< (ostream& os, Can acan);

};

#endif

//
//  implementation file can.cpp
//

#include "can.h"
#include <iostream>
using namespace std;

Can :: Can( double wt, double radius, double ht)
{
   empty_weight = wt;
   base_radius = radius;
   height = ht;
}

const double PI = 3.14159;
//
//  Given the volume (cm^3) of one gram of a product to be canned,
//  answer the question "How many whole grams of this product will fit
//  in this can?"
//
int Can :: capacity ( double volume ) const
{
   int result;

   result = PI * base_radius * base_radius * height / volume;

   return result;
}

istream& operator>> ( istream& is, Can& acan )
{
   is >> acan.empty_weight >> acan.base_radius >> acan.height;
   return is;
}

ostream& operator<< ( ostream& os, Can acan )
{
   os << "Can attributes:\n  empty weight: " << acan.empty_weight
      << " g\n  base radius: " << acan.base_radius <<
```

```
        " cm\n  height: " << acan.height << " cm\n\n";
   return os;
}

//
//  driver function
//

#include <iostream>
using namespace std;

#include <can.h>

int
main()
{
   Can can_1(10, 3.1, 7.0);
   cout << "Constructor-initialized can\n" << can_1;

   Can can_2;
   cout << "Enter these can attributes separated by spaces--\n"
     << "  empty weight in grams and base radius and height "
       << "in cm\n>>> ";
   cin >> can_2;
   cout << "Can input\n" << can_2;

   char again;
   double one_gram;

   do {
      cout << "Enter volume (cm^3) of one gram of a product \nto can>>> ";
        cin >> one_gram;
        cout << can_2.capacity(one_gram) << " whole grams of this "
           << "product will fit in can.\n";
        cout << "Do you want to enter another volume (y/n)?\n>>> ";
        cin >> again;
   } while ( again == 'y' );

   return 0;
}
```