

*The Art and Science of C*

Instructor's Manual

Eric S. Roberts  
Stanford University  
Stanford, California

## Section 1

# Converting from Pascal to C

From the late 1970s to the early 1990s, Pascal was the standard language for introductory computer science instruction throughout the United States and in much of the rest of the world as well. Over the last several years, however, many institutions have been moving away from Pascal as the language begins to show its age. Although some schools have taken a different direction, many—including Stanford—have adopted ANSI C as the language of instruction for the introductory course. Along with its object-oriented successor C++, ANSI C is the dominant language in the industry today. Teaching C early in the curriculum gives students additional preparation in the language they are most likely to use and makes for a smoother transition to C++ when that material appears later in the curriculum.

The reasons behind Stanford's decision to adopt C in the introductory course are outlined in the "To the Instructor" section of *The Art and Science of C* and discussed in detail in the paper "Using C in CS1: Evaluating the Stanford experience," which is included in Section 8 of this Instructor's Manual. The purpose of this section is not to recapitulate those reasons but to talk instead about strategies and tactics for making the change.

The two most important lessons to pass on from our experience at Stanford are the following:

1. *The transition will not be easy.* Anyone who suggests that changing the implementation language in an established curriculum is a simple task would be better employed marketing snake oil. The overall design of the introductory programming course depends to a surprising extent on the mechanics of the language system used to provide instruction; changing the language therefore has implications that affect the entire curriculum. The instructional staff needs to rethink how to present material, someone must rewrite the assignments, and the institutional structures supporting the introductory course must adapt to the new conditions.
2. *The advantages of making the change will be greater than anticipated.* At Stanford, our original goal in making the transition was to produce students in the new introductory course who programmed as effectively in C as their predecessors had in Pascal. Because C programming skills are more useful in subsequent courses, we believed that having the students attain the previous level of skill would be sufficient. What we've found is that students are much better programmers coming out of the revised course than they ever were before. They are more excited about the material, have worked harder to attain a level of mastery, and understand certain essential concepts—most notably modularity and data abstraction—much better than their predecessors did.

There are several things you can do to simplify the transition.

- *Take advantage of the benefits afforded by the library-based approach.* Because there are more details involved in learning C than in learning Pascal, it is important to simplify the conceptual process for students at the introductory level. In our experience, the most effective way to do so is to use the abstraction capabilities of the language to hide its complexity until the students are better able to understand it. The essential characteristics of the library-based approach are discussed in Section 2.
- *Prepare any affected staff for the transition.* Before making a change of this magnitude, it is important to make sure that the teaching staff responsible for the course—including TAs and anyone else who has direct contact with students in the course—have a chance to prepare for the transition. In the first year, we continued to teach a Pascal-based course while running a small, experimental C section. This phased introduction gave us time to refine the course materials and develop a cadre of support staff with the necessary expertise.

- *Recognize that the conversion will take time to stabilize.* One of the realities that we failed to understand when we first made the change at Stanford was how much we depend on the campus community as a whole to provide support for introductory teaching. In most years, students rely on older students in the dormitories for assistance. When we began to convert the introductory course, the students had no one to consult beyond the course staff, which was already overloaded trying to institute the changes. After a year, however, the informal support structures had been largely reestablished.

## Section 2

# The Library-Based Approach

Using *The Art and Science of C* makes it possible to teach C to introductory students without forcing them to assimilate all the details that make C difficult as a first language. For this purpose, the text uses a library-based approach that emphasizes the principle of abstraction.

Adopting the library-based approach has the following advantages:

1. *Libraries and modular development are covered in considerable detail early in the presentation.* Students using this text are exposed to the design and implementation of library packages much earlier than in most introductory courses. As a result, they complete the course with a much deeper understanding of not only the mechanics of libraries but also the associated concepts of abstraction and information hiding. By the middle of the first course, students understand the idea of an interface and appreciate the difference in perspective between the client and the implementation. This understanding makes it much easier for them to structure applications that adhere to the principles of modern software engineering.
2. *The libraries conceal many of the most difficult and confusing aspects of the language.* For the introductory student, the process of learning C is complicated by the fact that many of the concepts that one needs for simple programs are complex operations in the C domain. One of the classic examples is the `scanf` function, which requires students to use the `&` operator and to understand in detail the process of character I/O before they write their first program. Because they are using a library containing a much simpler collection of input functions, students do not have to confront these difficulties until much later in the course.
3. *The use of libraries makes it possible to present concepts in a logical, easily assimilated order.* The C programming language was designed for use by experienced programmers. As a result, understanding the underlying logic of the language often requires more detailed knowledge of programming concepts than the introductory student is likely to have. For example, C's approach to strings makes sense if you already understand arrays, which in turn make sense if you understand pointers. From the student's perspective, however, the internal dependencies among these concepts make learning much more difficult because strings are in fact conceptually simpler than the concepts on which strings are based. Introducing a library like `strlib.h` breaks the internal dependency and makes it possible to introduce the concepts in an order that makes them much easier for new students to grasp.
4. *Using libraries dramatically extends a student's ability to write exciting applications.* The graphics library introduced in Chapter 7 does more than introduce the notion of a library interface. By giving the students tools to draw simple figures on the computer screen, the `graphics.h` interface frees them from the sterile world of the text-based user interface and opens up the more exciting domain of computer graphics. Particularly now that most students have experience working with computers that offer graphical capabilities, being able to build graphical applications is essential to keeping student interest high.
5. *The text demonstrates the power of libraries by using them.* The libraries in the text serve not only as tools but also as objects of study in their own right. As the text reveals the implementation of the libraries, students gain a better understanding of how libraries work and how they can be used to control the complexity of a large software application.

The details and advantages of the library-based approach are considered more thoroughly in two papers included in Section 8 of this Instructor's Manual: "Using C in CS1: Evaluating the Stanford experience" and "A C-based graphics library for CS1."

## Section 3

# Obtaining Copies of the Libraries

To make it possible to teach ANSI C at the introductory level, *The Art and Science of C* is designed around several special libraries—collectively known as the **cslib** library—that hide much of the complexity associated with C until students are better equipped to understand how everything works. To use the text, you must have access to the **cslib** library on your own machine. Although the code for the **cslib** library is included in Appendix B of the text, it is much easier to obtain an electronic copy from the following FTP site:

`ftp://ftp.awl.com/cseng/authors/roberts/cs1-c`

With the exception of the graphics library presented in Chapter 7, the **cslib** library is entirely standard and requires nothing more than what is provided as part of ANSI C. Thus, if you do not intend to use the graphics library, it doesn't matter what version of **cslib** you get—they are all the same. However, because the mechanics of displaying screen images depend on the hardware and software configuration of each computer, you need an implementation of the graphics library that is appropriate to your computing environment.

To find the right version of the **cslib** library for your computing platform:

1. Check to see if your computing platform is any of the following:

- A Macintosh running THINK C or Symantec C++
- An IBM PC (or clone) running the Borland C/C++ compiler
- Any Unix system running the X Window System

For each of these platforms, there is a corresponding implementation of the libraries specific to that computing environment. To obtain it, you should connect to the appropriate subdirectory and follow the instructions in that **README** file. You should also check the **README** file on the FTP directory to see if new library releases have been made since this guide was published.

2. If you are using one of the hardware systems from the preceding list with a different compiler or operating system, consider purchasing the software necessary to make your platform correspond to one of the standard implementations. For example, if you own an IBM PC but use some compiler other than the Borland compiler, it might be worth installing the Borland compiler so that you can use all the features provided by the libraries, including interactive graphics. Although implementations for other platforms may be released in the future, adapting the graphics library is not a high-priority task now that there is at least one implementation for each of the major hardware platforms used in most universities.
3. If you cannot obtain the desired software or are using some other type of machine, get a copy of the standard version of the library code from the **standard** subdirectory.

This implementation defines the libraries in an ANSI-standard form but does not provide any on-screen graphics. The standard implementation of the graphics library instead writes a file containing a text representation of the image suitable for printing on any PostScript printer.

4. Although the standard library implementation is highly portable and works without change on every machine I've tried, there may be some systems that cannot support it. If you find that your system does not support the standard libraries for some reason, you should try the simplified version of the library, which provides the minimum level of support necessary to run the programs in the text.

### Subdirectory index

The top-level ftp directory contains no relevant files other than a **README** file. The files you need for the various libraries are collected in the following subdirectories, each of which is described in more detail in the sections that follow:

<b>simplified</b>	This subdirectory contains the simplest possible version of the library code and matches the code printed in Appendix B of the text (except for two minor changes introduced to increase
-------------------	--

portability). This implementation is machine-independent and requires no system support beyond the standard ANSI libraries. The implementation of the graphics library provides no actual screen display but instead writes a PostScript file containing the image. The simplified version of the **cslib** library should be used only if you are unable to get the standard version or one of the platform-specific implementations to work in your environment.

**standard** This subdirectory offers a more advanced version of the libraries than that used in the simplified package. One advantage of the standard version is that it is compatible with the programs in the forthcoming companion volume to *The Art and Science of C*, which will cover more advanced material focusing on data structures and algorithms. Most importantly, the standard version of the library includes a portable exception-handling package that makes it possible to use better error-recovery strategies. Using the standard version of the **cslib** library, as opposed to the simplified set, makes for a smoother transition to more advanced work.

Like the simplified package, the standard package implements the graphics library in a machine-independent way by writing a PostScript file. If you are working on a system for which there is a platform-specific implementation of the graphics library (these systems are enumerated later in this list), you should use that version instead. The standard version exists so that the fundamental libraries can be used on the widest possible set of platforms.

**unix-xwindows** Along with the facilities provided by the standard library package, this subdirectory contains an implementation of the graphics library designed for use with the X Window System, which is available on many computers that run the Unix operating system. The implementation is designed to work with several different versions of the Unix operating system and automatically configures itself to use the appropriate code for each.

**mac-think-c** This subdirectory contains an implementation of the graphics library for THINK C (or Symantec C/C++) on the Apple Macintosh. It also includes an interface that allows students to make a typescript of their computer session, which is a difficult operation with older versions of THINK C. To use this library, your Macintosh must be running System 7 of the operating system, and the THINK C compiler must be version 5.0 or later.

**pc-borland** This subdirectory contains two separate implementations of the graphics library for the Borland C/C++ compilers used on the IBM PC and its clones. The **dos** subdirectory provides a simple implementation of the basic graphics library on top of DOS. The **windows** subdirectory has a complete implementation of the graphics library that is compatible with Microsoft Windows. Except for the graphics library, the package is identical to the standard package. The **pc-borland** version of the libraries has been tested with Version 4 of the Borland C/C++ compiler. The same code may work with other versions of the compiler and associated software, but it is impossible to test it with every version of the compiler.

**documents** This subdirectory contains documentation on the libraries and the overall approach used in *The Art and Science of C*. See the **README** file in the documents directory for more information.

**programs** This subdirectory contains electronic copies of all sample programs used in the text. If you are teaching a course using this text and want solutions to the exercises, please contact your Addison-Wesley representative.

#### Notes and disclaimers

The **cslib** libraries are in the public domain and may be freely copied and distributed, although they remain under development. No warranties are made concerning their correctness or stability, and no user support is offered.

## Section 4

### Designing a Syllabus

The structure of the syllabus for an introductory programming course depends significantly on many factors specific to the institution. Stanford, for example, is on the quarter system, which means that the course material must fit into a ten-week quarter. Schools that follow a more traditional semester calendar have more class meetings, although there is nonetheless considerable variability in the length of the term even among those institutions. Because of this variability, it is difficult to design a syllabus—or a text for that matter—that is precisely appropriate to the needs of all institutions.

Beyond the variations in the academic calendar, different schools also tend to cover material at different speeds. At Stanford, we move at a relatively fast pace that may not be appropriate for all institutions. To ensure that students do not get left behind in the process, we provide an extensive infrastructure that supports the more traditional lecture component of the course. Every quarter, we hire a large number of advanced undergraduates to lead sections and provide general course assistance, which includes staffing the main computing cluster 70 hours a week. (This program is described in the paper “Using undergraduates as teaching assistants in introductory programming courses,” which appears in Section 8 of this Instructor's Manual.) It is the fact that we have a large, easily accessible staff that makes it possible for us to move as quickly as we do, covering essentially a semester's worth of material in a ten-week quarter.

The “To the Instructor” section of the text describes each of the chapters and offers various suggestions about how to order the material. The next two pages show a typical syllabus from Stanford's CS106A course, which encompasses the basic CS1 material from Curriculum '78. As the syllabus shows, we cover the entire the text with the exception of the topics in Chapter 17, which are presented the following quarter in CS106B.

The syllabus also shows several additions to the curriculum beyond the material in *The Art and Science of C*. The major extensions in the Stanford course are that

- *We start the quarter with a “preprogramming” exercise.* At the beginning of the first programming course, it is much more important for students to learn about programming as a general problem-solving methodology than it is for them to learn all the syntactic details of the specific language used in the course. Over the years, we have found that the best way to focus the students' attention on the more important questions of algorithmic design is devote the first three days to Richard Pattis's fine book *Karel the Robot: A Gentle Introduction to the Art of Programming*. Having used this book for many years when the course was taught in Pascal, we were pleased that it continued to be very effective after the transition to C.
- *We include a unit on a simulated computer called CSim.* Particularly when students begin to learn about arrays and pointers in C, it is essential for them to have a model of how memory and addressing work. To give students more of a hands-on sense of how the computer operates, we start this section by introducing a locally developed simulated computer called CSim. Like Donald Knuth's MIX machine described in *The Art of Computer Programming*, the CSim machine has a restricted set of capabilities and a simplified instruction set that are nonetheless sufficient to illustrate basic principles of how computers work at the machine-language level. Copies of the CSim simulator and accompanying descriptive materials will become publicly available sometime in 1995; in the meantime, it is sufficient to rely on the discussion of memory organization in Chapter 11.
- *We include discussion of the social impact of computing.* Computers play an increasingly important role in all aspects of our society and, as a result, computer professionals have a greater responsibility to ensure that their work benefits society. At Stanford, we emphasize the importance of good software engineering throughout the introductory course and talk about the risks associated with the software development process.

# Syllabus for CS106A

## Autumn Quarter 1993–94

Monday	Wednesday	Friday
	September 29  Administration What is programming? Introduction to Karel  Read: Pattis, Chapters 1–2	October 1  Defining new instructions Control structures in Karel  Read: Pattis, Sections 3.1–3.5, 4.1–4.5, 5.1–5.4
4  Program decomposition Stepwise refinement  Read: Pattis, Sections 3.8–3.9, 5.5	6  Introduction to C C program structure Simple input and output  Read: Roberts, Chapter 1, Sections 2.1–2.3 Due: Assignment #1	8  Values, variables, and types Arithmetic expressions Precedence  Read: Sections 2.4–2.5
11  Problem solving Programming idioms Loops: <b>for</b> and <b>while</b>  Read: Chapter 3 Due: Karel contest entry	13  More problem solving Conditional testing The <b>if</b> and <b>switch</b> statement  Read: Chapter 4	15  Functions and procedures Writing your own functions Using libraries  Read: Chapter 5 Due: Assignment #2
18  A simple graphics library Defining tools Decomposition in C  Read: Sections 7.1–7.3	20  More on the graphics library The contents of an interface Procedural abstraction  Read: Section 7.4	22  Random numbers Defining an interface  Read: Chapter 8 Due: Assignment #3
25  Algorithms  Read: Chapter 6	27  Debugging strategies  Read: Handout on debugging Due: Assignment #4	29  Midterm Exam (90 minutes, open book)



Monday	Wednesday	Friday
November 1  Character data Strings as abstract types The <code>strlib.h</code> library  Read: Chapter 9 Due: Graphics contest entry	3  More on strings Modular development  Read: Section 10.1	5  More modular development Local versus global data  Read: Sections 10.2–10.3
8  Opening the black box Introduction to CSim The function of a compiler  Read: Handout on CSim	10  Introduction to arrays  Read: Chapter 11 Due: Assignment #5	12  Sorting Selection sort Evaluating an algorithm  Read: Chapter 12
15  The concept of a pointer Pointer arithmetic  Read: Sections 13.1–13.3	17  Pointers and arrays Dynamic memory allocation  Read: Sections 13.4–13.5 Due: Assignment #6	19  Implementing <code>strlib.h</code> The <code>string.h</code> interface  Read: Chapter 14
22  Text files The standard I/O library  Read: Sections 15.1–15.3	24  Text files and <code>stdio.h</code> (continued)  Read: Sections 15.4–15.5 Due: Assignment #7	26  Thanksgiving (no class)
29  Introduction to records  Read: Sections 16.1–16.3	December 1  Records and data abstraction Pointers to records Allocating records  Read: Sections 16.4–16.5	3  Data structures Software engineering  Read: Sections 16.6 Due: Assignment #8A
6  The future of computing  Read: Handout on future visions Due: C contest entry	8  Social implications of computing  Read: Handout on social implications	10  Review for final  Read: Practice final Due: Assignment #8B

## Section 5

# Assignments

Programming assignments are the heart of most introductory computer science courses. Although students can pick up theoretical concepts and basic strategies in class, they learn very little about programming from the lectures and text alone. Programming is a learn-by-doing enterprise. For most students, understanding comes from applying their knowledge in the context of a programming problem. Because of the centrality of the act of programming to the learning experience, it is important to put considerable thought into the design of your assignments.

It is also important to realize that programming assignments serve two distinct purposes.

1. *Assignments make it possible to reinforce particular concepts.* As you go through the chapters in the text, students need practice with the specific concepts each chapter introduces. Thus, when you cover the chapter on functions, you want an assignment that gives the student practice using functions and functional decomposition.
2. *Assignments capture the students' interest by allowing them to solve challenging problems.* For most students, learning to program is an empowering experience. As they begin to solve real problems, the programming process becomes its own reward, encouraging them to work even harder on the material.

In most cases, these two goals tend to drive the design of assignments in different directions. To reinforce concepts, you need to design assignments that are relatively narrow in their focus. To generate excitement, you want assignments that are both much larger and more wide-ranging in their coverage.

As the assignments in this section indicate, we tend to combine the two styles. Of the eight assignments during the quarter, several concentrate on specific topics from the text and ask students to solve problems selected from the programming exercises. Others, particularly assignments #3 and #8, give students a chance to write larger programs that require them to integrate many different concepts in a single project. I believe that such a mix works very well.

The following hints may prove useful in designing assignments of your own:

- *Invest some time in assignment design.* It's important to think carefully about what you want your assignments to achieve, and then to evaluate them to see how well they work. Most of the best programming assignments evolve over time. Brown University professor Andries VanDam, the winner of the ACM Outstanding Educator Award of 1994, has argued that it takes four years to get an assignment right.
- *Make the assignments fun.* Getting students excited about the material gives them the incentive to learn it on their own. If students find a problem dull, they will work much less hard on it.
- *Use graphics in assignments.* Because students today have grown up with computers that offer extensive graphical capabilities, intergating graphics into the design of your assignments inevitably increases student enthusiasm. The graphics library introduced in Chapter 7 makes it possible to perform simple graphical operations on most of the common computing platforms.
- *Give good students a chance to excel.* At most institutions, the introductory course attracts students with widely varying backgrounds and aptitudes. Teaching at too high a level risks losing much of the class; teaching at too low a level bores the better students. At Stanford, we address this problem by assigning periodic programming contests that give the better students a chance to show what they can do without changing the course requirements for everyone else.
- *Make sure grading does not interfere with the educational value of assignments.* No matter how well the assignments are designed, much of their value is lost if students spend more time thinking about their grade than they do about the concepts. The paper entitled "Using undergraduates as teaching assistants in introductory programming courses" describes a grading system designed to reduce grade competition; a copy of that paper is included in Section 8.

## Assignment 1—Getting Started

### Part I—Get an electronic mail account

Over the last fifteen years, computer science professionals and students have tended to rely more and more on electronic mail (usually called e-mail for short) for their communications. CS106A provides you with an introduction to programming, which is the core technological skill required for computer science. Along the way, we would like to introduce you to state-of-the-art computing technology. To this end, we require all students in CS106A to have an e-mail account.

Since having an e-mail account is of negligible value if you ignore its existence, we will expect you to read your mail regularly through the quarter and encourage you to use this medium to communicate with the course staff. We will use it to communicate with you, so you should be sure to check your mailbox at least once every two days. Having an e-mail account is also an extremely fast (and free!) way to communicate with friends in other schools, so there are other advantages for getting on-line as well.

*The remainder of this section should describe how to get a mail account at your institution.*

### Part II—Hunt the Wumpus

As soon as your mail account becomes active, the next step in this assignment is to go to one of the campus Macintoshes and play a game called *Hunt the Wumpus*. One of the earliest computer games, Hunt the Wumpus was developed by the People's Computer Company in Silicon Valley about 20 years ago. In recent years, with all the exciting computer games that have come along, the wumpus has become nearly extinct, but as a game hunting the wumpus still has considerable charm. For one thing, it can be played at several different levels of strategy. Children run around in the cave without much forethought, happily losing a bunch of games just to get the occasional thrill of a successful game. People who think about the game a little more discover that it can involve some subtle strategies.

Another advantage of playing the wumpus game early in the quarter is that doing so gives us something to shoot for, so to speak. In the final assignment of the quarter, your mission will be to implement the wumpus game in its entirety. The programs you use today will become those you write tomorrow.

The wumpus game displays its own rules, and your job is to play it until you win. When you win your first game, the program will display your own secret password to congratulate you on your victory. Remember your password for Part III.

### Part III—Send mail to your section leader

As soon as you complete your foray against the wumpus, send an e-mail message to your instructor. Your e-mail message should include your name, the password from the wumpus game, and a brief description of what you hope to get out of this course.

### Part IV—Solve some simple Karel programs

*The final part of the first assignment at Stanford consists of several simple programs for Karel the Robot. If you use Karel or some similar preprogramming exercise in your course, you should include it as part of the first assignment.*

## Assignment 2 — Simple C Programs

This assignment gives you a chance to solve several simple programs using C. All the problems are taken from the text.

1. Chapter 2, exercise 3, page 55.
2. Chapter 3, exercise 9, page 95.
3. Chapter 3, exercise 11, page 95.
4. Chapter 3, exercise 14, page 96.
5. Chapter 4, exercise 2, page 134.
6. Chapter 4, exercise 7, page 134.

---

Assignment #2 covers the material in Chapters 2–4 and reinforces the students' knowledge of expressions, control statements, `printf`, and the `stdio.h` facilities for input. Particularly if you want to vary exercises from term to term, you can substitute other exercises in place of these, although you should strive to retain a balance of topics. It is also important to find some question that allows the student to develop a nontrivial solution strategy. Finding the largest number in a list (Chapter 3, exercise 14) is my favorite exercise of this form, although there are others that work almost as well.

Because the problems in Assignment #2 are taken entirely from the text, you can find the solutions in Section 7.

## Assignment 3 — Decomposition

For this assignment, your job is to write a program that emphasizes decomposition in the style of the `calendar.c` program from Chapter 5. Before you write any code, it is important to sit down and think about what needs to be done and how to organize your solution strategy into a well-structured program.

### A check-writing program

Chapter 3 includes a program that might help you balance your checkbook. Given that you have such a program, it might be useful to have the computer generate the actual checks as well. For example, if you need to write a check for \$1,234.56 to the Acme Mortgage Company, your program should be able to write out a check that looks like this:

15-Oct-93	1001
Pay to the order of: Acme Mortgage Company	\$ 1234.56
Amount: one thousand two hundred thirty-four and 56/100	
<hr/>	

The main program should allow the user to write several checks in a row. At the beginning of the program, the user should be asked for

- the date, which is entered as a string
- the starting check number

For all checks written in the current session, the program should use the same date and should number the checks consecutively from a starting point specified by the user.

For each check, the program should request

- the amount of the check in numeric form
- the name of payee

After reading in this data, the program should write out a diagram of the check, filling in each of the fields in the appropriate format. When the user enters 0 as the amount of a check, the program should interpret that value as a sentinel indicating the end of the session.

A complete sample run of the program that writes out three checks is shown on the next page.

```
Enter starting check number: 1001
Enter the date: 15-Oct-93
Enter the amounts and payees for each check.
Use 0 to exit from the program.
Amount of check (0 to stop): 1234.56
Pay to the order of: Acme Mortgage Company

-----

                                15-Oct-93          1001

Pay to the order of: Acme Mortgage Company          $ 1234.56
Amount: one thousand two hundred thirty-four and 56/100

-----

Amount of check: 17775
Pay to the order of: Stanford University

-----

                                15-Oct-93          1002

Pay to the order of: Stanford University          $ 17775.00
Amount: seventeen thousand seven hundred seventy-five and 00/100

-----

Amount of check: 22.95
Pay to the order of: Acme Long Distance

-----

                                15-Oct-93          1003

Pay to the order of: Acme Long Distance          $ 22.95
Amount: twenty-two and 95/100

-----

Amount of check: 0
End of program
```

When you write your program, the output should appear in precisely the format shown on the previous page. For example, your program is responsible for displaying the two lines of hyphens that separate each check from the input data and for aligning the fields so that the same fields appear in the same places on each check. In the sample run shown, each of the dividing lines is 66 hyphens long. The other fields are arranged to align against one side of the check or the other, except for the date, which ends in column 55. The signature line is composed of 25 underscore characters; the underscore appears above the hyphen on the Macintosh keyboard. For background on how to align these fields, you should read section 3.5 in the text, which covers everything you need.

The hard part of the program, and the part that will take both the most thinking and the most decomposition, is translating the numeric check amount into its English form. The largest check that your program needs to handle is \$999,999.99. Your program must break that number into pieces and translate each piece into words through a series of procedure calls. In many cases, the procedures will have a form similar to

```
void PrintOneDigit(int d)
{
    switch (d) {
        case 0: printf("zero"); break;
        case 1: printf("one"); break;
        case 2: printf("two"); break;
        case 3: printf("three"); break;
        case 4: printf("four"); break;
        case 5: printf("five"); break;
        case 6: printf("six"); break;
        case 7: printf("seven"); break;
        case 8: printf("eight"); break;
        case 9: printf("nine"); break;
        default: Error("Illegal call to PrintOneDigit");
    }
}
```

The interesting question, however, is how to assemble the different procedures into a program that handles all parts of the number. Be on the lookout for functions that you can reuse. For example, displaying the first three digits of a six-digit amount (the number of thousands) is closely related to displaying the last three digits, and you should avoid duplicating code as much as possible.

---

The purpose of Assignment #3 is to get the students thinking about functions and decomposition. To drive home the importance of top-down design and the reuse of common functions, the project has to be large enough to support several layers of decomposition. Our students find this assignment challenging but doable.

## Solution to Assignment 3

```
/*
 * File: check.c
 * -----
 * This program writes a check for any amount less than
 * a million dollars.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constants
 * -----
 * Sentinel          -- Value used to stop
 * MaxCheck          -- Largest possible check
 * CheckWidth        -- width of check
 * SignatureWidth    -- width of signature
 */

#define Sentinel      0
#define MaxCheck      999999.99
#define CheckWidth    66
#define SignatureWidth 25

/* Function prototypes */

void PrintCheck(int num, string date, string payee, double amount);
void PrintBorder(void);
void PrintDateLine(string date, int num);
void PrintPayeeLine(string payee, double amount);
void PrintEnglishAmountLine(double amount);
void PrintSignatureLine(void);
void PrintThreeDigits(int ddd);
void PrintTwoDigits(int dd);
void PrintDecade(int dd);
void PrintTeens(int dd);
void PrintOneDigit(int d);
void PrintNCopies(int n, string s);
int Round(double x);
```



```
/* Main program */

main()
{
    int num;
    double amount;
    string date, payee;

    printf("Enter starting check number: ");
    num = GetInteger();
    printf("Enter the date: ");
    date = GetLine();
    printf("Enter the amounts and payees for each check.\n");
    printf("Use %g to exit from the program.\n", (double) Sentinel);
    while (TRUE) {
        printf("Amount of check (%g to stop): ", (double) Sentinel);
        amount = GetReal();
        if (Round(amount * 100) == Round(Sentinel * 100)) break;
        if (amount > 0 && amount <= MaxCheck) {
            printf("Pay to the order of: ");
            payee = GetLine();
            PrintCheck(num, date, payee, amount);
            num++;
        } else {
            printf("That amount is too large.\n");
        }
    }
}

/*
 * Function: PrintCheck
 * Usage: PrintCheck(num, date, payee, amount);
 * -----
 * This function displays a check using the argument values.
 */

void PrintCheck(int num, string date, string payee, double amount)
{
    PrintBorder();
    PrintDateLine(date, num);
    printf("\n");
    PrintPayeeLine(payee, amount);
    PrintEnglishAmountLine(amount);
    printf("\n\n");
    PrintSignatureLine();
    printf("\n");
    PrintBorder();
}

/*
 * Function: PrintBorder
 * Usage: PrintBorder();
 * -----
 * This function is used to display the top and bottom border.
 */

void PrintBorder(void)
{
    printf("\n");
    PrintNCopies(CheckWidth, "-");
    printf("\n\n");
}
```

```
/*
 * Function: PrintDateLine
 * Usage: PrintDateLine();
 * -----
 * This function is used to display the line containing the
 * date and the check number.
 */

void PrintDateLine(string date, int num)
{
    printf("%55s %10d\n", date, num);
}

/*
 * Function: PrintPayeeLine
 * Usage: PrintPayeeLine();
 * -----
 * This function is used to display the line containing the
 * payee and the amount in figures.
 */

void PrintPayeeLine(string payee, double amount)
{
    printf("Pay to the order of: %-34.34s $%9.2f\n", payee, amount);
}

/*
 * Function: PrintEnglishAmountLine
 * Usage: PrintEnglishAmountLine();
 * -----
 * This function is used to translate the value in figures into
 * an amount in English.
 */

void PrintEnglishAmountLine(double amount)
{
    int thousands, last3, cents;
    double leftover;

    printf("Amount: ");
    thousands = amount / 1000;
    leftover = amount - 1000 * thousands;
    last3 = leftover;
    cents = Round(100 * (leftover - last3));
    if (thousands != 0) {
        PrintThreeDigits(thousands);
        printf(" thousand");
        if (last3 != 0) {
            printf(" ");
            PrintThreeDigits(last3);
        }
    } else {
        PrintThreeDigits(last3);
    }
    if (thousands != 0 || last3 != 0) printf(" and ");
    printf("%02d/100\n", cents);
}
```

```
/*
 * Function: PrintSignatureLine
 * Usage: PrintSignatureLine();
 * -----
 * This function is used to display the signature line.
 */

void PrintSignatureLine(void)
{
    PrintNCopies(CheckWidth - SignatureWidth, " ");
    PrintNCopies(SignatureWidth, "_");
}

/*
 * Function: PrintThreeDigits
 * Usage: PrintThreeDigits(ddd);
 * -----
 * This function is used to display a three-digit number
 * in its English form and is used for both the thousands
 * and the last three digits of the number.
 */

void PrintThreeDigits(int ddd)
{
    int hundreds;

    hundreds = ddd / 100;
    if (hundreds != 0) {
        PrintOneDigit(hundreds);
        printf(" hundred ");
    }
    PrintTwoDigits(ddd % 100);
}

/*
 * Function: PrintTwoDigits
 * Usage: PrintTwoDigits(dd);
 * -----
 * This function is used to display a two-digit number
 * in its English form.
 */

void PrintTwoDigits(int dd)
{
    int tens, units;

    tens = dd / 10;
    units = dd % 10;
    if (tens == 0) {
        PrintOneDigit(units);
    } else if (tens == 1) {
        PrintTeens(dd);
    } else {
        PrintDecade(tens * 10);
        if (units != 0) {
            printf("-");
            PrintOneDigit(units);
        }
    }
}
```

```
/*
 * Function: PrintDecade
 * Usage: PrintDecade(dd);
 * -----
 * This function takes a two-digit number ending in 0 and
 * prints out the English name of that number. For example
 * PrintDecade(20) prints "twenty". The cases 00 and 10
 * must be handled specially; PrintDecade generates an error
 * if passed these values.
 */

void PrintDecade(int dd)
{
    switch (dd) {
        case 20: printf("twenty"); break;
        case 30: printf("thirty"); break;
        case 40: printf("forty"); break;
        case 50: printf("fifty"); break;
        case 60: printf("sixty"); break;
        case 70: printf("seventy"); break;
        case 80: printf("eighty"); break;
        case 90: printf("ninety"); break;
        default: Error("Illegal call to PrintDecade");
    }
}

/*
 * Function: PrintTeens
 * Usage: PrintTeens(dd);
 * -----
 * This function takes a two-digit number between 10 and 19
 * and prints its name. These values must be handled as
 * special cases.
 */

void PrintTeens(int dd)
{
    switch (dd) {
        case 10: printf("ten"); break;
        case 11: printf("eleven"); break;
        case 12: printf("twelve"); break;
        case 13: printf("thirteen"); break;
        case 14: printf("fourteen"); break;
        case 15: printf("fifteen"); break;
        case 16: printf("sixteen"); break;
        case 17: printf("seventeen"); break;
        case 18: printf("eighteen"); break;
        case 19: printf("nineteen"); break;
        default: Error("Illegal call to PrintTeens");
    }
}
```

```
/*
 * Function: PrintSingleDigit
 * Usage: PrintSingleDigit(d);
 * -----
 * This function prints the English equivalent of a single
 * digit.
 */

void PrintOneDigit(int d)
{
    switch (d) {
        case 0: printf("no"); break;
        case 1: printf("one"); break;
        case 2: printf("two"); break;
        case 3: printf("three"); break;
        case 4: printf("four"); break;
        case 5: printf("five"); break;
        case 6: printf("six"); break;
        case 7: printf("seven"); break;
        case 8: printf("eight"); break;
        case 9: printf("nine"); break;
        default: Error("Illegal call to PrintOneDigit");
    }
}

/*
 * Function: PrintNCopies
 * Usage: PrintNCopies(n, s);
 * -----
 * This function prints n copies of the string s. It is used
 * for generating the borders, the signature line, and the internal
 * spacing.
 */

void PrintNCopies(int n, string s)
{
    int i;

    for (i = 0; i < n; i++) {
        printf("%s", s);
    }
}

/*
 * Function: Round
 * Usage: n = Round(x);
 * -----
 * This function returns the nearest integer to x.
 */

int Round(double x)
{
    if (x < 0) {
        return ((int) (x - 0.5));
    } else {
        return ((int) (x + 0.5));
    }
}
```

## Assignment 4 — Graphics and Random Numbers

For this assignment, your job is to solve three simple exercises from the text that use the graphics and random number libraries:

1. Chapter 7, exercise 6, page 254.
2. Chapter 7, exercise 7, page 255.
3. Chapter 8, exercise 12, page 299.

---

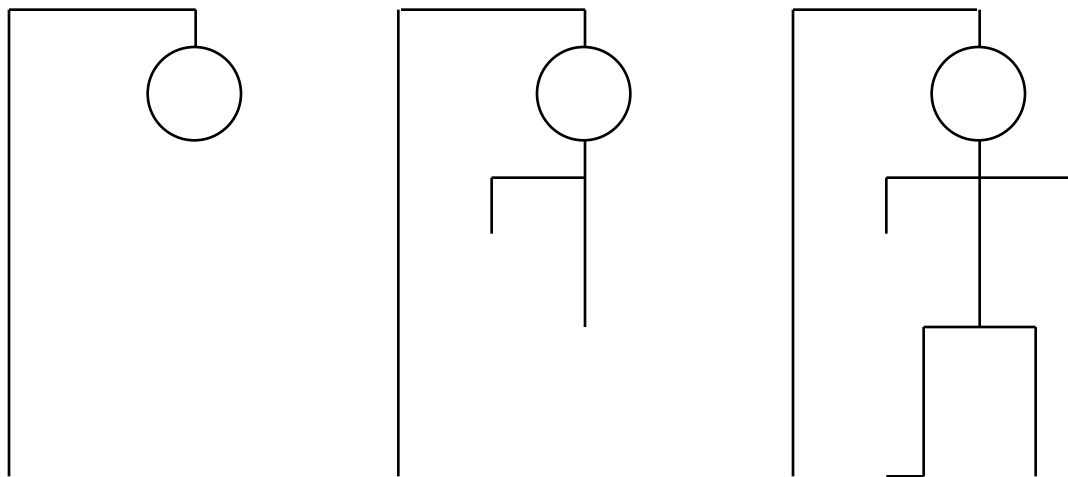
Assignment #4 covers the material in Chapters 7–8. As with Assignment #2, it is easy to vary this assignment from term to term by substituting other problems from the text. Because these problems are taken entirely from the text, you can find the solutions in Section 7.

## Assignment 5 — String Manipulation

For Assignment #5, your mission is to write a program that plays the game of Hangman. The assignment serves two purposes. First, it is designed to give you some practice writing programs that manipulate strings using the facilities described in Chapter 9. Second, it gives you a chance to write a program that is larger than those you have written for most previous assignments. As you discovered with the check-writing program, large programs require more up-front organization than small programs, and it is important that you begin to get experience with larger programs at this point in the course. Because of the additional organizational work and the fact that there are more details to manage, doubling the size of a program usually ends up having that program require more than twice as much work to get it going. The key piece of advice, therefore, is to start early.

When it plays Hangman, the computer first selects a secret word at random from a list built into the program. The program then displays a row of dashes—one for each letter in the secret word—and asks the user to guess a letter. If the player guesses a letter that is in the word, the word is redisplayed with all instances of that letter shown in the correct positions, along with any letters correctly guessed on previous turns. If the letter does not appear in the word, the player is charged with an incorrect guess. The player keeps guessing letters until either (1) all the letters in the word have been correctly guessed or (2) the player has made eight incorrect guesses. Two sample runs that illustrate the play of the game are shown on the next two pages.

When it is played by children, the real fascination (a somewhat morbid one, I suppose) of Hangman comes from the fact that incorrect guesses are recorded by drawing an evolving picture of the player being hanged on a scaffold. For each incorrect guess, a new part of a stick-figure body—first the head, then the body, then each arm, each leg, and finally each foot—is added to the scaffold until the hanging is complete. For example, the following three diagrams show the drawing after the first incorrect guess (just the head), the third (the head, body, and left arm), and the diagram at the tragic end of a losing game:



In order to write the program that plays Hangman, you should design and test your program in two parts. First, get the interactive part working without any graphics at all. Once that part is done and working, go back and add the code to draw the picture using the graphics library described in Chapter 7. The two parts of the assignment are explained after the sample runs.

```
Welcome to Hangman!
The word now looks like this: -----
You have 8 guesses left.
Your guess: A
There are no A's in the word.
The word now looks like this: -----
You have 7 guesses left.
Your guess: E
There are no E's in the word.
The word now looks like this: -----
You have 6 guesses left.
Your guess: I
There are no I's in the word.
The word now looks like this: -----
You have 5 guesses left.
Your guess: O
There are no O's in the word.
The word now looks like this: -----
You have 4 guesses left.
Your guess: U
That guess is correct.
The word now looks like this: -U---
You have 4 guesses left.
Your guess: S
There are no S's in the word.
The word now looks like this: -U---
You have 3 guesses left.
Your guess: T
There are no T's in the word.
The word now looks like this: -U---
You have 2 guesses left.
Your guess: R
There are no R's in the word.
The word now looks like this: -U---
You have only one guess left.
Your guess: N
There are no N's in the word.
You're completely hung.
The word was: FUZZY
You lose.
```

Sample Run #1



```
Welcome to Hangman!
The word now looks like this: -----
You have 8 guesses left.
Your guess: a
There are no A's in the word.
The word now looks like this: -----
You have 7 guesses left.
Your guess: e
That guess is correct.
The word now looks like this: -----E-
You have 7 guesses left.
Your guess: i
There are no I's in the word.
The word now looks like this: -----E-
You have 6 guesses left.
Your guess: o
That guess is correct.
The word now looks like this: -O----E-
You have 6 guesses left.
Your guess: u
That guess is correct.
The word now looks like this: -O--U-E-
You have 6 guesses left.
Your guess: s
There are no S's in the word.
The word now looks like this: -O--U-E-
You have 5 guesses left.
Your guess: t
That guess is correct.
The word now looks like this: -O--UTE-
You have 5 guesses left.
Your guess: r
That guess is correct.
The word now looks like this: -O--UTER
You have 5 guesses left.
Your guess: c
That guess is correct.
The word now looks like this: CO--UTER
You have 5 guesses left.
Your guess: m
That guess is correct.
The word now looks like this: COM-UTER
You have 5 guesses left.
Your guess: p
That guess is correct.
You guessed the word: COMPUTER
You win.
```

Sample Run #2

## Part I

In the first part of this assignment, your job is to write a program that handles the user interaction component of the game—everything except the graphical display. To solve the problem, your program must be able to

- Choose a random secret word from a list
- Keep track of the user's partially guessed word, which begins as a list of dashes and then is updated as correct letters are guessed
- Implement the control structure of the interaction (looping for each guess, detecting the end of the game, and so forth)
- Manage the details (keep track of the number of guesses, display the right messages, and so on)

The only operation that is beyond your current knowledge is the process of choosing a random word. To solve that part of the problem, you should use the `randword.h` interface, which is supplied in the **Assignment 5** folder on the public server. The `randword.h` interface contains two functions: `InitDictionary` and `RandomWord`. The `InitDictionary` function initializes the dictionary and should be called before you make any other calls on the package. The `RandomWord` function takes no arguments and returns a randomly chosen English word.

For the rest of Part I, you need to use string manipulation along the lines shown in the Pig Latin program from Chapter 10. Think first about what the overall structure of the program is and how to break the program down into separate functions. Figure out what variables you will need and how information flows back and forth between the different functions that make up the complete solution.

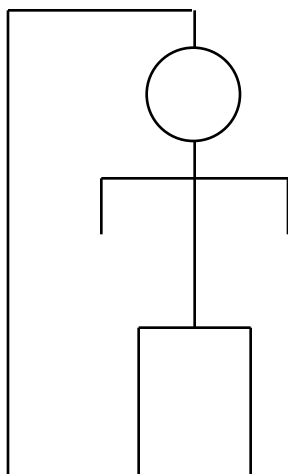
Your program should implement the following features:

- Guesses should be accepted either in lower or upper case, even though all letters in the secret words are written in upper case.
- If the player guesses something other than a single letter, your program should provide the opportunity to guess again.
- If the player guesses a correct letter more than once, your program can simply do nothing. Guessing an incorrect letter a second time should be counted as another wrong guess. (In either case, these interpretations are the easiest way to handle the situation, and your program will probably do the right thing even if you don't think about these cases in detail.)

Remember to finish Part I before moving on to Part II. Part II is arguably more fun, but it is critically important to develop large programs in manageable stages.

## Part II

For Part II, your task is simply to extend the program you have already written so that it now keeps track of the Hangman graphical display. Everything you need to draw a simple picture is available in the graphics library. Your task is simply to determine the best way to integrate the new part of the program into the existing structure. The complete picture used in the sample application appears at the top of the next page.



The scaffold is drawn initially before the game begins, and then the parts are added in the following order: head, body, left arm, right arm, left leg, right leg, left foot, right foot. The constants that define the parameters for this picture are shown below:

```

/*
 * Constants
 * -----
 * The following constants define the dimensions of the
 * Hangman picture. All values are given in inches.
 *
 * ScaffoldX      x coordinate of scaffold base
 * ScaffoldY      y coordinate of scaffold base
 * ScaffoldHeight height of vertical scaffold section
 * BeamLength     length of the top cross beam
 * RopeLength     length of the rope between head and beam
 * HeadRadius     radius of the head
 * BodyLength     length of the stick-figure body
 * ArmOffset      distance from head to shoulders
 * UpperArmLength horizontal length of upper part of arm
 * LowerArmLength vertical length of lower part of arm
 * HipWidth       horizontal length between body and leg
 * LegLength      vertical length of the leg
 * FootLength     length of each foot
 */

#define ScaffoldX      0.5
#define ScaffoldY      0.5
#define ScaffoldHeight 2.5
#define BeamLength     1.0
#define RopeLength     0.25
#define HeadRadius     0.25
#define BodyLength     1.0
#define ArmOffset      0.2
#define UpperArmLength 0.5
#define LowerArmLength 0.3
#define HipWidth       0.25
#define LegLength      0.75
#define FootLength     0.2

```

To give you some experience working with modules, an important part of this assignment is to implement Parts I and II as separate modules. The main module should be the `hangman.c` program you wrote in Part I augmented by the appropriate calls to a new module that handles the graphical operations. The interface to the new module should be called `hangpict.h` and must export a function for drawing the scaffold at the beginning of the game and another function for drawing the next body part. The detailed design of the interface, however, is up to you.

## Solutions to Assignment 5

This program is similar to the hangman problem in Chapter 14 with two exceptions: (1) it uses the `strlib.h` interface instead of `string.h` and (2) it includes the graphical module as an explicit part of the assignment.

```
/*
 * File: hangman.c
 * -----
 * This program plays a game of Hangman.
 */

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

#include "genlib.h"
#include "strlib.h"
#include "simpio.h"
#include "random.h"
#include "graphics.h"
#include "randword.h"
#include "hangpict.h"

/* Private function prototypes */

static void GiveInstructions(void);
static void PlayHangmanGame(void);
static char ReadUsersGuess(void);
static string UpdatePattern(char ch, string pattern, string secret);
static bool IsContainedIn(char ch, string word);
static string ConcatNCopies(int n, string str);

/* Implementation */

main()
{
    InitGraphics();
    Randomize();
    InitDictionary();
    InitHangmanPicture();
    GiveInstructions();
    PlayHangmanGame();
}

/*
 * Function: GiveInstructions
 * Usage: GiveInstructions();
 * -----
 * This function prints the rules to Hangman.
 */

static void GiveInstructions(void)
{
    printf("Welcome to Hangman!\n");
    printf("I will guess a secret word. On each turn, you guess\n");
    printf("a letter. If the letter is in the secret word, I\n");
    printf("will show you where it appears; if not, a part of\n");
    printf("your body gets strung up on the scaffold. The\n");
    printf("object is to guess the word before you are hung.\n");
}
```

```
/*
 * Function: PlayHangmanGame
 * Usage: PlayHangmanGame();
 * -----
 * This function plays one game of Hangman with the user. The program
 * first chooses a secret word from the list stored in the dictionary
 * and then creates a pattern word of the same length using only hyphens.
 * As the game proceeds, the pattern characters are replaced by the
 * actual letters in the secret word. If the word is guessed or the
 * player exceeds the guess limit, the game is over.
 */

static void PlayHangmanGame(void)
{
    string secret, pattern;
    int guessesLeft;
    char guess;

    secret = RandomWord();
    pattern = ConcatNCopies(StringLength(secret), "-");
    while ((guessesLeft = GuessesRemaining()) > 0
        && !StringEqual(pattern, secret)) {
        printf("The word now looks like this: %s\n", pattern);
        if (guessesLeft == 1) {
            printf("You have only one guess left.\n");
        } else {
            printf("You have %d guesses left.\n", guessesLeft);
        }
        guess = ReadUsersGuess();
        if (IsContainedIn(guess, secret)) {
            printf("That guess is correct.\n");
            pattern = UpdatePattern(guess, pattern, secret);
        } else {
            printf("There are no %c's in the word.\n", guess);
            DrawNextBodyPart();
        }
    }
    if (guessesLeft == 0) {
        printf("You're completely hung.\n");
        printf("The word was: %s\n", secret);
        printf("You lose.\n");
    } else {
        printf("You guessed the word: %s\n", secret);
        printf("You win.\n");
    }
}
```

```
/*
 * Function: ReadUserGuess
 * Usage: ch = ReadUserGuess();
 * -----
 * Requests a guess from the user and checks it for legality.
 * Legal guesses are single letters, which are converted to
 * uppercase before being returned. If a guess is not legal,
 * the user is asked to supply a new guess.
 */

static char ReadUsersGuess(void)
{
    string line;

    while (TRUE) {
        printf("Your guess: ");
        line = GetLine();
        if (StringLength(line) != 1) {
            printf("Your guess must be a single character.");
        } else if (!isalpha(IthChar(line, 0))) {
            printf("Your guess must be a letter.");
        } else {
            return (toupper(IthChar(line, 0)));
        }
        printf(" Try again.\n");
    }
}

/*
 * Function: UpdatePattern
 * Usage: pattern = UpdatePattern(ch, pattern, secret);
 * -----
 * This function goes through the pattern string and creates
 * a new string that contains the same characters except for
 * those positions in which ch occurs in the secret word. In
 * those positions, the new string contains ch. This operation
 * encapsulates the act of guessing a letter in Hangman.
 */

static string UpdatePattern(char ch, string pattern, string secret)
{
    int i;
    string result;

    result = "";
    for (i = 0; i < StringLength(pattern); i++) {
        if (secret[i] == ch) {
            result = Concat(result, CharToString(ch));
        } else {
            result = Concat(result,
                            CharToString(IthChar(pattern, i)));
        }
    }
    return (result);
}
```

```
/*
 * Function: IsContainedIn
 * Usage: if (IsContainedIn(ch, word)) . . .
 * -----
 * This function returns TRUE if the character ch appears in the string
 * word.
 */

static bool IsContainedIn(char ch, string word)
{
    return (FindChar(ch, word, 0) != -1);
}

/*
 * Function: ConcatNCopies
 * Usage: newstr = ConcatNCopies(n, str);
 * -----
 * This function creates a new string consisting of n copies of str
 * concatenated together. (ConcatNCopies appears in Chapter 9.)
 */

static string ConcatNCopies(int n, string str)
{
    string result;
    int i;

    result = "";
    for (i = 0; i < n; i++) {
        result = Concat(result, str);
    }
    return (result);
}
```

```
/*
 * File: hangpict.h
 * -----
 * This file contains the interface to a module that draws the
 * scaffold and handles the other graphics required for the
 * Hangman program.
 */

#ifndef _hangpict_h
#define _hangpict_h

/*
 * Function: InitHangmanPicture
 * Usage: InitHangmanPicture();
 * -----
 * This function initializes the hangpict module and draws the
 * empty scaffold. The client is responsible for calling InitGraphics
 * before calling this function. This design makes it easier to
 * ensure that the initialization of the graphics library occurs
 * at the very beginning of main.
 */

void InitHangmanPicture(void);

/*
 * Function: DrawNextBodyPart
 * Usage: DrawNextBodyPart();
 * -----
 * This function draws the next body part in the Hangman figure.
 */

void DrawNextBodyPart(void);

/*
 * Function: GuessesRemaining
 * Usage: guesses = GuessesRemaining();
 * -----
 * This function returns the number of guesses remaining. That
 * information is kept inside the hangpict module because the
 * number of guesses is determined by the number of body parts.
 * Using this design, changing the number of legal guesses
 * requires changes only to the hangpict.c implementation, not
 * the main program.
 */

int GuessesRemaining(void);

#endif
```



```

/*
 * File: hangpict.c
 * -----
 * This file implements the hangpict.c interface.
 */

#include <stdio.h>
#include "genlib.h"
#include "graphics.h"
#include "hangpict.h"

/*
 * Constants
 * -----
 * The following constants define the dimensions of the
 * Hangman picture. All values are given in inches.
 */
/*
 * NBodyParts      The number of body parts
 * ScaffoldX       x coordinate of scaffold base
 * ScaffoldY       y coordinate of scaffold base
 * ScaffoldHeight  height of vertical scaffold section
 * BeamLength      length of the top cross beam
 * RopeLength      length of the rope between head and beam
 * HeadRadius      radius of the head
 * BodyLength      length of the stick-figure body
 * ArmOffset       distance from head to shoulders
 * UpperArmLength  horizontal length of upper part of arm
 * LowerArmLength  vertical length of lower part of arm
 * HipWidth        horizontal length between body and leg
 * LegLength       vertical length of the leg
 * FootLength      length of each foot
 */

#define NBodyParts      8
#define ScaffoldX       0.5
#define ScaffoldY       0.5
#define ScaffoldHeight  2.5
#define BeamLength      1.0
#define RopeLength      0.25
#define HeadRadius      0.25
#define BodyLength      1.0
#define ArmOffset       0.2
#define UpperArmLength  0.5
#define LowerArmLength  0.3
#define HipWidth        0.25
#define LegLength       0.75
#define FootLength      0.2

/*
 * Constants: Left, Right
 * -----
 * Since the body is symmetrical, the program can be shortened
 * by using the same code for the left and right sides of the
 * figure, passing along a direction indicator with a value of
 * +1 or -1. If you multiply the relevant x-coordinate values
 * by this factor, the body part will show up on the left or
 * right side of the body, as appropriate.
 */

#define Left -1
#define Right 1

```

```
/*
 * Private variables
 * -----
 * topx, topy -- The coordinates of the top of the head.  Saving these
 *              values simplifies the drawing.
 * guessesLeft -- The number of guesses left.
 */

static double topx, topy;
static int guessesLeft;

/* Private function prototypes */

static void DrawHead(void);
static void DrawBody(void);
static void DrawArm(int direction);
static void DrawLeg(int direction);
static void DrawFoot(int direction);
static void DrawCenteredCircle(double x, double y, double r);

/* Exported functions */

void InitHangmanPicture(void)
{
    MovePen(ScaffoldX, ScaffoldY);
    DrawLine(0, ScaffoldHeight);
    DrawLine(BeamLength, 0);
    DrawLine(0, -RopeLength);
    topx = GetCurrentX();
    topy = GetCurrentY();
    guessesLeft = NBodyParts;
}

void DrawNextBodyPart(void)
{
    switch (guessesLeft) {
        case 8: DrawHead(); break;
        case 7: DrawBody(); break;
        case 6: DrawArm(Left); break;
        case 5: DrawArm(Right); break;
        case 4: DrawLeg(Left); break;
        case 3: DrawLeg(Right); break;
        case 2: DrawFoot(Left); break;
        case 1: DrawFoot(Right); break;
        default: Error("Illegal body part");
    }
    guessesLeft--;
}

int GuessesRemaining(void)
{
    return (guessesLeft);
}
```

```
/* Private functions */

static void DrawHead(void)
{
    DrawCenteredCircle(topx, topy - HeadRadius, HeadRadius);
}

static void DrawBody(void)
{
    MovePen(topx, topy - 2 * HeadRadius);
    DrawLine(0, -BodyLength);
}

static void DrawArm(int direction)
{
    MovePen(topx, topy - 2 * HeadRadius - ArmOffset);
    DrawLine(direction * UpperArmLength, 0);
    DrawLine(0, -LowerArmLength);
}

static void DrawLeg(int direction)
{
    MovePen(topx, topy - 2 * HeadRadius - BodyLength);
    DrawLine(direction * HipWidth, 0);
    DrawLine(0, -LegLength);
}

static void DrawFoot(int direction)
{
    MovePen(topx + direction * HipWidth,
            topy - 2 * HeadRadius - BodyLength - LegLength);
    DrawLine(direction * FootLength, 0);
}

/*
 * Function: DrawCenteredCircle
 * Usage: DrawCenteredCircle(x, y, r);
 * -----
 * This function draws a circle of radius r with its
 * center at (x, y). This function appears in Chapter 7.
 */

static void DrawCenteredCircle(double x, double y, double r)
{
    MovePen(x + r, y);
    DrawArc(r, 0, 360);
}
```

## Assignment 6 — CSim

At this point in the quarter, the Stanford course digresses from the material in the text to cover CSim, a simulated computer system that helps give students a mental model of memory and instruction execution. The CSim simulator is not yet available for export and was therefore not included in *The Art and Science of C*. The problems that follow illustrate the level of the presentation.

1. Write a program for the CSim simulator that duplicates as closely as possible the operation of the following C program:

```
main()
{
    int i;

    for (i = 10; i >= 0; i--) {
        printf("%d\n", i);
    }
}
```

2. In mathematics, there is a famous sequence of numbers called the *Fibonacci sequence* after the 13th-century Italian mathematician Leonardo Fibonacci. The first two terms in this sequence are 0 and 1, and every subsequent term is the sum of the preceding two. Thus, the first several numbers in the Fibonacci sequence are

$$\begin{array}{rcl}
 F_0 & = & 0 \\
 F_1 & = & 1 \\
 F_2 & = & 1 \quad (0 + 1) \\
 F_3 & = & 2 \quad (1 + 1) \\
 F_4 & = & 3 \quad (1 + 2) \\
 F_5 & = & 5 \quad (2 + 3) \\
 F_6 & = & 8 \quad (3 + 5)
 \end{array}$$

The Fibonacci function can be implemented in C as follows:

```
int Fib(int n)
{
    int t1, t2, t3, i;

    t1 = 0;
    t2 = 1;
    for (i = 0; i < n; i++) {
        t3 = t1 + t2;
        t1 = t2;
        t2 = t3;
    }
    return (t1);
}
```

Write a CSim program that defines a separate Fibonacci function **Fib** which takes its argument in the **AC** and returns its result there as well. Test your function by writing a simple main program that reads in a number **n** and writes out **Fib(n)**.

## Assignment 7 — Arrays

This assignment gives you the opportunity to work with arrays and sorting.

### Problem 1

Solve Chapter 11, exercise 8 on page 420.

### Problem 2

Programs that simulate card games usually have to simulate the operation of shuffling the deck. Like sorting, shuffling is a process that involves rearranging the elements of an array. Algorithmically, the only difference between sorting and shuffling is how you select elements. When you sort an array using selection sort, you choose the smallest element in the rest of the array on each cycle of the loop. When you shuffle an array, you choose a random element.

Write a function **Shuffle** that shuffles an array of strings. To test the **Shuffle** function, write a program that

1. Declares an array with 52 elements, each of which are strings.
2. Fills the elements of that array with strings representing standard playing cards. Each card is represented by a string consisting of a rank (A, K, Q, J, 10, 9, 8, 7, 6, 5, 4, 3, 2) concatenated with a single letter representing a suit (C, D, H, S). Thus, the queen of spades is represented by the string "QS". The function **IntegerToString** described in Chapter 9 (page 302) will probably come in handy here.
3. Shuffles the array using the **Shuffle** function.
4. Deals a bridge hand by copying the first 13 cards from the deck to a separate array.
5. Sorts the 13 cards in the hand so that the cards are in descending order by suit. When the hand is sorted, all the spades come first, followed by the hearts, the diamonds, and finally the clubs. Within each suit, the cards should be listed in the order given in step 2: first the ace, then the king, then the queen, and so on down to the two. Note that this step requires an operation that is pretty much the same as the **sort** function in the text. The only differences are that (1) the array elements are strings and (2) the comparison operation is slightly more complicated. In all other ways, your program should follow exactly the same structure.
6. Displays the 13 cards in the hand on a single line.

The following is a sample run of the program:

```
Hand: AS 6S AH 10H 7H 4H KD 8D 5D 4D 3D 2D AC
```

## Solutions to Assignment #7

1. The solution to the histogram problem appears in Section 7 (Solutions to Exercises).

2.

```
/*
 * File: shuffle.c
 * -----
 * This file implements the shuffling program.
 */

#include <stdio.h>
#include "genlib.h"
#include "strlib.h"
#include "random.h"

/*
 * Constants
 * -----
 * NCards      The number of cards in the deck (52)
 * NCardsInHand The number of cards in a hand
 */

#define NCards      52
#define NCardsInHand 13

/* Private function prototypes */

static void InitDeck(string deck[]);
static void Shuffle(string array[], int n);
static void DealHand(string hand[], int n, string deck[], int ccard);
static void SortHand(string hand[], int n);
static int FindLargestCard(string hand[], int low, int high);
static bool CardIsLarger(string c1, string c2);
static void SwapStringElements(string array[], int p1, int p2);
static void DisplayHand(string hand[], int n);
static int Rank(string card);
static char Suit(string card);

/* Main program */

main()
{
    string deck[NCards], hand[NCardsInHand];

    Randomize();
    InitDeck(deck);
    Shuffle(deck, NCards);
    DealHand(hand, NCardsInHand, deck, 0);
    SortHand(hand, NCardsInHand);
    DisplayHand(hand, NCardsInHand);
}
```

```
/*
 * Function: InitDeck
 * Usage: InitDeck(deck);
 * -----
 * This function initializes the deck, which must be allocated by the
 * caller as an array of 52 strings.
 */

static void InitDeck(string deck[])
{
    string rank;
    int i, cp;

    cp = 0;
    for (i = 1; i <= 13; i++) {
        switch (i) {
            case 1: rank = "A"; break;
            case 11: rank = "J"; break;
            case 12: rank = "Q"; break;
            case 13: rank = "K"; break;
            default: rank = IntegerToString(i); break;
        }
        deck[cp++] = Concat(rank, "C");
        deck[cp++] = Concat(rank, "D");
        deck[cp++] = Concat(rank, "H");
        deck[cp++] = Concat(rank, "S");
    }
}

/*
 * Function: Shuffle
 * Usage: Shuffle(array, n);
 * -----
 * This function randomly shuffles the first n elements in array.
 * The algorithm is exactly the same as that for sorting, except
 * that the new first element is chosen randomly instead of by
 * finding the largest element.
 */

static void Shuffle(string array[], int n)
{
    int lh, rh;

    for (lh = 0; lh < n; lh++) {
        rh = RandomInteger(lh, n - 1);
        SwapStringElements(array, lh, rh);
    }
}
```

```
/*
 * Function: DealHand
 * Usage: DealHand(hand, n, deck, ccard);
 * -----
 * This function deals n cards from the deck into the array hand. The
 * ccard parameter indicates the current card number in the deck, which
 * makes it possible to use this function to deal several hands.
 */

static void DealHand(string hand[], int n, string deck[], int ccard)
{
    int i;

    for (i = 0; i < n; i++) {
        hand[i] = deck[i + ccard];
    }
}

/*
 * Function: SortHand
 * Usage: SortHand(hand, n);
 * -----
 * The SortHand function sorts the hand into rank order by suits.
 * The only differences between this function and that used in the
 * Sort function from Chapter 12 are (1) the change in base type
 * and (2) a different comparison function.
 */

static void SortHand(string hand[], int n)
{
    int lh, rh;

    for (lh = 0; lh < n-1; lh++) {
        rh = FindLargestCard(hand, lh, n-1);
        SwapStringElements(hand, lh, rh);
    }
}

/*
 * Function: FindLargestCard
 * Usage: index = FindLargestCard(hand, low, high);
 * -----
 * Returns the index of the largest value in array
 * between the indices low and high, inclusive.
 * It operates by keeping track of the index of the
 * largest card so far in the variable largest.
 */

static int FindLargestCard(string hand[], int low, int high)
{
    int i, largest;

    largest = low;
    for (i = low; i <= high; i++) {
        if (CardIsLarger(hand[i], hand[largest])) largest = i;
    }
    return (largest);
}
```



```
/*
 * Function: CardIsLarger
 * Usage: if (CardIsLarger(c1, c2)) . . .
 * -----
 * This function returns TRUE if the card represented by
 * the string c1 is larger than (i.e., comes earlier in the
 * sorted hand than) the card represented by c2. The suits
 * are checked first; only if the suits match are the ranks
 * compared. Note that the ace is handled as a special case.
 */

static bool CardIsLarger(string c1, string c2)
{
    int suit1, suit2;
    int rank1, rank2;

    suit1 = Suit(c1);
    suit2 = Suit(c2);
    if (suit1 != suit2) return (suit1 > suit2);
    rank1 = Rank(c1);
    rank2 = Rank(c2);
    if (rank1 == 1) rank1 = 14;
    if (rank2 == 1) rank2 = 14;
    return (rank1 > rank2);
}

/*
 * Function: SwapStringElements
 * Usage: SwapStringElements(array, p1, p2);
 * -----
 * This function swaps the elements in array at index positions p1 and p2.
 */

static void SwapStringElements(string array[], int p1, int p2)
{
    string tmp;

    tmp = array[p1];
    array[p1] = array[p2];
    array[p2] = tmp;
}

/*
 * Function: DisplayHand
 * Usage: DisplayHand(hand, n);
 * -----
 * This function displays the n cards in the array indicated by hand.
 */

static void DisplayHand(string hand[], int n)
{
    int i;

    printf("Hand: ");
    for (i = 0; i < n; i++) {
        if (i > 0) printf(" ");
        printf("%s", hand[i]);
    }
    printf("\n");
}
```

```
/*
 * Function: Rank
 * Usage: rank = Rank(c);
 * -----
 * This function returns the rank of the card as an integer
 * between 1 (ace) and 13 (king).
 */

static int Rank(string card)
{
    char ch;

    ch = IthChar(card, 0);
    switch (ch) {
        case 'A': return (1);
        case '1': return (10);
        case 'J': return (11);
        case 'Q': return (12);
        case 'K': return (13);
        default: return (ch - '0');
    }
}

/*
 * Function: Suit
 * Usage: suit = Suit(c);
 * -----
 * This function returns the suit of the card c.
 */

static char Suit(string card)
{
    return (IthChar(card, StringLength(card) - 1));
}
```

## Assignment 8 — Implementing Hunt the Wumpus

There are two purposes to this assignment. First, it gives you a chance to work with arrays, records, and files—all of which provide additional experience in working with data structures. Second, this assignment gives you an opportunity to build a significant program consisting of many independent pieces. As you do so, you will have to test the individual pieces as the program grows; you will almost certainly fail if you try to build the entire program all at once.

The assignment has two parts, each of which is described later in this handout. Part I is a written assignment in which you design the data structure for the wumpus cave and then write a single function that uses that structure to answer a question. Part II of the assignment is the completed wumpus game.

### The rules to Hunt the Wumpus

You all played Hunt the Wumpus as part of your first assignment and should have a good idea of how it works. The rules, which the program displays if you ask for instructions, are as follows:

The wumpus lives in a cave of 20 rooms. Each room has tunnels leading to other rooms elsewhere in the cave.

Caution is necessary in exploring the cave. Three of the rooms contain bottomless pits. If you walk into one of these rooms, you fall into the pit and lose. Three other rooms are home to a species of super bats, strong enough to lift a human. If you enter one of these rooms, the bats will pick you up and randomly drop you in some other room within the cave, where you could fall into a pit, or you might run into . . . the wumpus.

The wumpus is the most fearsome denizen of the underground caverns. It is not bothered by the other hazards because it has sucker feet and is too heavy for a bat to lift. Its favorite diet is adventurers. Fortunately for you, however, the wumpus is usually asleep. If you walk into its room or shoot an arrow anywhere in the cave, the wumpus wakes up and may decide either to stay where it is or wander to an adjacent room. In either case, if the wumpus ends up in the same room as you, it finishes you off as a light snack.

On each turn, you can decide either to move along a corridor to an adjacent room or shoot one of your five magic arrows. (If you enter a number in response to the “Move or shoot” question, the program assumes you want to move to that room.) If you decide to shoot an arrow, you must then give a list of room numbers, one per line, ending with a blank line. The arrows can go around corners and even backtrack, but must always move to an adjacent room. If your list of room numbers would send the arrow to a room that is not connected to its current location, it picks a random tunnel and keeps going. If your arrow hits the wumpus, you win. If it hits you, you lose.

To help you navigate, your computer guide provides you with several warnings. If you are one room away from a bottomless pit, the program lets you know by saying “I feel a draft.” If you are one room away from bats, it tells you “Bats nearby.” And if you are one or two rooms away from the wumpus in any direction, it says “I smell a wumpus.”

Happy Hunting!

Your mission in this assignment is to write a C program that plays this game.

## Part I—Designing the data structure

When you begin to solve a programming problem on this scale, one of your first tasks is to design the data structure. In the case of the wumpus game, your program needs to store the information that represents the state of the game, which includes such data as how the rooms are connected, where the various hazards are, where the player is, how many arrows are left, and so forth.

In designing a data structure, it is usually best to model it as closely on its real-world counterparts as you can. Doing so makes it easier for programmers to understand the structure by allowing them to rely on their own intuition. For example, the wumpus game takes place in a cave that consists of 20 rooms. A good design for your data structure might therefore include a type **CaveT** that represents the cave and a type **RoomT** that represents an individual room within the cave. The game program would create one **CaveT** value and 20 **RoomT** values, each of which would contain information pertinent to that aspect of the wumpus world. For example, each **RoomT** data structure must contain a list of the other rooms to which it is connected. The **CaveT** structure, on the other hand, is more appropriate for data pertaining to the cave as a whole.

The details of how the **CaveT** and **RoomT** types are represented and what internal values they contain are your responsibility. Making diagrams helps, as does thinking about the operations that the wumpus implementation will require. In particular, thinking about the use of the structures is helpful in determining whether you want to work with records directly or with pointers to records.

## Deliverables for Part I

Part I of this assignment has two subparts, each of which calls for a paper design rather than an actual implementation.

1. Design all the type definitions necessary to represent the wumpus cave and the information represented therein. The **CaveT** structure should contain all the information you need to maintain the state of the game. Thus you should be able to declare a variable **cave** of type **CaveT** in the main program and have that structure be sufficient to store all the data the program needs at any point. For Part I, you do not need to write the code to initialize this structure—all you need are the structure definitions.
2. Using your data structure as defined in the preceding step, write the code necessary to implement the predicate function with the following prototype:

```
bool ISmellAWumpus(caveT cave);
```

The effect of this function is to examine the cave data structure and determine whether the player can smell the wumpus, which happens whenever the player is one or two rooms away from where the wumpus lurks. Since you have not yet written code to initialize the data structure, you cannot immediately test your function, and you should simply write it out on paper as if this were a question on an exam.

## Part II—The implementation

To a large extent, the deliverables for this assignment are determined by the rules of the wumpus game. You know how to play wumpus; your task is therefore to implement a program that lets someone play the game according to those rules. There are, however, a couple of aspects of the program that bear additional discussion.

### Displaying the instructions

Although it would be easy enough to display the instructions for the wumpus game using individual **printf** statements, it is more convenient, when instructions are this long, to store the instructions in a data file and then display the contents of that file on the terminal when the player requests instructions. The instructions for playing wumpus are included in the file **rules.dat**, which is in the **Assignment #8** folder.

The fact that the instructions are long, however, brings up a new problem. If you simply write out all the lines in the file, the instructions will scroll off the screen before the player has a chance to read them. To avoid this problem, the **rules.dat** file contains lines of the form

```
<WAIT>
```

after each screenful of information. Your function to write out the instructions must look for this line as it reads the `rules.dat` file. When this line comes up, the function should wait for the user to type a blank line before proceeding.

### Initializing the cave

One of the hardest parts of the wumpus program is initializing the cave data structure. The wumpus implementation you played as part of Assignment #1 used a random process to create a suitable cave. Building a random cave, however, turns out to be tricky. For the assignment, you should instead read in the connections for the cave (which mathematicians would call its *topology*) from a data file. The **Assignment #8** folder contains nine different files, named `topology-1.dat` through `topology-9.dat`, that specify reasonable topologies for the wumpus cave. Each file has 20 lines that contain a room number, a colon, and then the numbers of the three rooms to which that room connects.

For example, the contents of `topology-1.dat` are as follows:

```
1: 6 14 16
2: 3 7 18
3: 2 16 20
4: 6 18 19
5: 8 9 11
6: 1 4 15
7: 2 12 19
8: 5 10 13
9: 5 11 17
10: 8 14 16
11: 5 9 18
12: 7 14 15
13: 8 15 20
14: 1 10 12
15: 6 12 13
16: 1 3 10
17: 9 19 20
18: 2 4 11
19: 4 7 17
20: 3 13 17
```

This file shows that room 1 connects to 6, 14, and 16; room 2 connects to 3, 7, and 18; and so on.

To generate the cave, your program must choose one of the nine topology files at random and then read in the connections from that file. To complete the initialization, your program also needs to select starting locations for the player, the wumpus, and the other hazards.

### Additional requirements

In addition to the specific points discussed above, your program should obey the following implementation requirements:

- All user responses should be accepted in either upper or lower case. Moreover, the first letter should be sufficient as an abbreviation for the complete words **yes**, **no**, **move**, or **shoot**.
- As a convenience, the user should be able to type a room number in response to the “Move or shoot” query, which is then interpreted as a move command to that room.
- Your program should allow the user to play many games and should keep track of the won-lost score for the session.

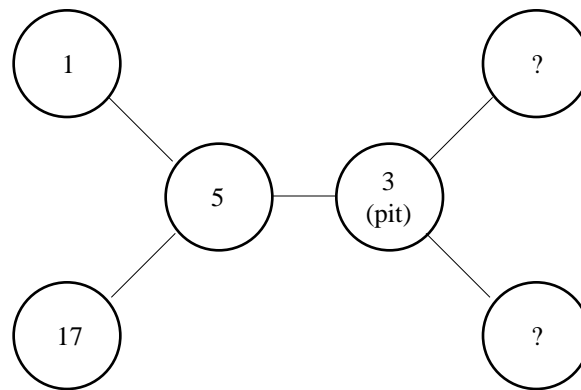
### Further clarifications of the rules

Please make sure that your program correctly implements the following aspects of the wumpus rules:

- The adventurer, the wumpus, the three bats, and the three pits all start the game in distinct rooms.

- The wumpus wakes up (a) if the adventurer moves into its room or (b) after an arrow has completed its flight path. The wumpus then randomly chooses either to stay in the same room (50% probability) or to move randomly to an adjacent room. The wumpus only eats the adventurer if they end up in the same room. Because the wumpus “has sucker feet and is too heavy for a bat to lift” it can go into rooms containing the other hazards. As soon as it moves (or stays put) the wumpus goes back to sleep.
- If bats pick up the adventurer, they fly to a random room anywhere in the cave, drop the adventurer, and then fly back to their initial room.
- Each arrow always travels through one room for each of the room numbers listed by the user. If there is a tunnel leading from the arrow's current room to the next room in the input list, the arrow goes to that room. If not, the arrow randomly picks one of the three rooms connected to the current room and goes there. This process is repeated as long as there are additional rooms in the user's list.

This behavior can actually be useful as a tactical ploy. Consider the following room diagram:



In room 5, you both smell the wumpus and feel a draft, but you have been in both rooms 1 and 17 without detecting any of these hazards. You therefore know that room 3 contains a pit and that the wumpus must be in one of two rooms labeled with question marks on the other side of room 3. Even though you don't even know the numbers of those rooms, you can give yourself a reasonable chance of hitting the wumpus by backing up to room 1 and sending an arrow on the path 5, 3, 3, 3, 3.

The first two rooms are certainly connected along a path, so the arrow makes it unimpeded to room 3. From room 3, however, there is certainly no tunnel to room 3, so the arrow then picks one of the connected rooms at random. It might be room 5, but it might also be one of the unknown rooms, where it might hit the wumpus. If it doesn't, the next room in the list must be okay, because there must be a path back to the room from which the arrow just came. The last room in the path again tries a random corridor and gives you one more chance to hit the wumpus. The procedure is completely safe and, even if you miss, may get the wumpus to move closer.

If you have trouble interpreting the guidelines for this assignment, you should run the wumpus application and see how it behaves.

## Solutions to Assignment 8

```
/*
 * File: wumpus.c
 * -----
 * This program plays Hunt the Wumpus. The program itself is
 * divided into two modules to reduce each component of the
 * program to a more manageable size:
 *   wumpus.c -- main module containing most of the game
 *   cave.c   -- data structures and initialization of the cave
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"
#include "strlib.h"
#include "random.h"
#include "cave.h"

/*
 * Constants
 * -----
 * RuleFile -- File containing rules
 */

#define RuleFile "rules.dat"

/* Private function prototypes */

static void WelcomePlayer(void);
static void GiveInstructions(void);
static void PlayWumpusGame(caveT cave);
static void TakeOneTurn(caveT cave);
static void DescribeRoom(caveT cave);
static void ExecutePlayerCommand(caveT cave);
static void MakeMove(caveT cave, int room);
static void ShootArrow(caveT cave);
static bool IsConnected(caveT cave, int r1, int r2);
static bool AdjacentToBat(caveT cave);
static bool AdjacentToPit(caveT cave);
static bool AdjacentToWumpus(caveT cave, int r1);
static bool ISmellAWumpus(caveT cave);
static void PrintStandings(caveT cave);
static bool GetYesOrNo(string prompt);

main()
{
    caveT cave;
    int i;

    Randomize();
    WelcomePlayer();
    cave = NewCave();
    while (TRUE) {
        PlayWumpusGame(cave);
        if (!GetYesOrNo("Would you like to play again? ")) break;
    }
    PrintStandings(cave);
}
```

```
/*
 * Function: WelcomePlayer
 * Usage: WelcomePlayer();
 * -----
 * This function welcomes the player and checks whether the
 * player needs instructions.
 */

static void WelcomePlayer(void)
{
    printf("Welcome to Hunt the Wumpus!\n");
    if (GetYesOrNo("Would you like instructions? ")) GiveInstructions();
    printf("\n");
}

/*
 * Function: GiveInstructions
 * Usage: GiveInstructions();
 * -----
 * This function reads the rules file and prints its contents as
 * instructions to the user. When the function encounters a line
 * in the file of the form <WAIT>, it waits for the user to press
 * the return key before proceeding.
 */

static void GiveInstructions(void)
{
    FILE *infile;
    string line;

    infile = fopen(RuleFile, "r");
    if (infile == NULL) {
        Error("The rules.dat file must be in the project folder");
    }
    while ((line = ReadLine(infile)) != NULL) {
        if (StringEqual(line, "<WAIT>")) {
            printf("<press return to continue> ");
            (void) GetLine();
        } else {
            printf("%s\n", line);
        }
    }
}

/*
 * Function: PlayWumpusGame
 * Usage: PlayWumpusGame(cave);
 * -----
 * This function plays a single wumpus game. The cave structure must
 * be allocated by the caller.
 */

static void PlayWumpusGame(caveT cave)
{
    printf("You're descending into the cave of the wumpus.\n");
    printf("Down . . .\n");
    printf("  Down . . .\n");
    printf("    Down . . .\n\n");
    SetupCave(cave);
    cave->done = FALSE;
    while (!cave->done) TakeOneTurn(cave);
}
```



```
/*
 * Function: TakeOneTurn
 * Usage: TakeOneTurn(cave);
 * -----
 * This function performs the operations necessary to take one turn
 * for the player.
 */

static void TakeOneTurn(caveT cave)
{
    printf("You are in room %d\n", cave->player);
    if (cave->rooms[cave->player].bat) {
        printf("There are bats in your room.\n");
        cave->player = RandomInteger(1, NRooms);
    } else if (cave->rooms[cave->player].pit) {
        printf("You fell into a pit.\n");
        cave->playerLosses++;
        cave->done = TRUE;
    } else if (cave->player == cave->wumpus) {
        printf("You were eaten by the wumpus.\n");
        cave->playerLosses++;
        cave->done = TRUE;
    } else {
        DescribeRoom(cave);
        ExecutePlayerCommand(cave);
    }
}

/*
 * Function: DescribeRoom
 * Usage: DescribeRoom(cave);
 * -----
 * This function displays the connections from the player's current
 * room.
 */

void DescribeRoom(caveT cave)
{
    int i;

    printf("There are tunnels to");
    for (i = 0; i < 3; i++) {
        printf(" %d", cave->rooms[cave->player].connections[i]);
    }
    printf("\n");
    if (AdjacentToBat(cave)) printf("Bats nearby\n");
    if (AdjacentToPit(cave)) printf("I feel a draft\n");
    if (ISmellAWumpus(cave)) printf("I smell a wumpus\n");
}
```

```
/*
 * Function: ExecutePlayerCommand
 * Usage: ExecutePlayerCommand(cave);
 * -----
 * This function reads a command from the player and performs the
 * requested operation.
 */

static void ExecutePlayerCommand(caveT cave)
{
    string response;

    printf("Move or shoot? ");
    response = GetLine();
    switch (response[0]) {
        case 'm': case 'M':
            printf("Which room? ");
            MakeMove(cave, GetInteger());
            break;
        case 's': case 'S':
            ShootArrow(cave);
            break;
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            MakeMove(cave, StringToInteger(response));
            break;
        default:
            printf("I don't understand that.\n");
    }
}

/*
 * Function: MakeMove
 * Usage: MakeMove(cave, room);
 * -----
 * This function attempts to move the player to the specified room.
 * If there is no connection to the requested room, the function
 * displays a message to that effect and does not move the player.
 */

static void MakeMove(caveT cave, int room)
{
    if (IsConnected(cave, cave->player, room)) {
        cave->player = room;
    } else {
        printf("There is no path to that room\n");
    }
}
```

```
/*
 * Function: ShootArrow
 * Usage: ShootArrow(cave);
 * -----
 * This function is called when the player has decided to shoot
 * an arrow. The implementation takes care of requesting room
 * numbers for the arrow's flight path and checking to see what
 * happens along the way.
 */

static void ShootArrow(caveT cave)
{
    string response;
    int arrow, room, count, tunnel;

    printf("Give list of rooms, terminated by a blank line.\n");
    arrow = cave->player;
    for (count = 0; count < MaxPath; count++) {
        response = GetLine();
        if (StringLength(response) == 0) break;
        room = StringToInteger(response);
        if (IsConnected(cave, arrow, room)) {
            arrow = room;
        } else {
            arrow = cave->rooms[arrow].connections[RandomInteger(0,2)];
        }
        if (arrow == cave->player) {
            printf("You shot yourself.\n");
            cave->playerLosses++;
            cave->done = TRUE;
            return;
        }
        if (arrow == cave->wumpus) {
            printf("You shot the wumpus.\n");
            cave->playerWins++;
            cave->done = TRUE;
            return;
        }
    }
    cave->arrows--;
    if (cave->arrows == 0) {
        printf("You ran out of arrows.\n");
        cave->playerLosses++;
        cave->done = TRUE;
        return;
    }
    if (RandomChance(0.5)) {
        tunnel = RandomInteger(0, 2);
        cave->wumpus = cave->rooms[cave->wumpus].connections[tunnel];
    }
}
```

```
/*
 * Function: IsConnected
 * Usage: if (IsConnected(cave, r1, r2)) . . .
 * -----
 * This function returns TRUE if there is a connection between rooms
 * r1 and r2 in the cave.
 */

static bool IsConnected(caveT cave, int r1, int r2)
{
    int i;

    for (i = 0; i < 3; i++) {
        if (cave->rooms[r1].connections[i] == r2) return (TRUE);
    }
    return (FALSE);
}

/*
 * Function: AdjacentToBat
 * Usage: if (AdjacentToBat(cave)) . . .
 * -----
 * This function returns TRUE if there is a bat in any of the
 * rooms connected to the player's current room.
 */

static bool AdjacentToBat(caveT cave)
{
    int i, r;

    for (i = 0; i < 3; i++) {
        r = cave->rooms[cave->player].connections[i];
        if (cave->rooms[r].bat) return (TRUE);
    }
    return (FALSE);
}

/*
 * Function: AdjacentToPit
 * Usage: if (AdjacentToPit(cave)) . . .
 * -----
 * This function returns TRUE if there is a pit in any of the
 * rooms connected to the player's current room.
 */

static bool AdjacentToPit(caveT cave)
{
    int i, r;

    for (i = 0; i < 3; i++) {
        r = cave->rooms[cave->player].connections[i];
        if (cave->rooms[r].pit) return (TRUE);
    }
    return (FALSE);
}
```

```
/*
 * Function: AdjacentToWumpus
 * Usage: if (AdjacentToWumpus(cave, room)) . . .
 * -----
 * This function returns TRUE if the specified room is adjacent to
 * to the room where the wumpus is.
 */

static bool AdjacentToWumpus(caveT cave, int r1)
{
    int i, r2;

    for (i = 0; i < 3; i++) {
        r2 = cave->rooms[r1].connections[i];
        if (cave->wumpus == r2) return (TRUE);
    }
    return (FALSE);
}

/*
 * Function: ISmellAWumpus
 * Usage: if (ISmellAWumpus(cave)) . . .
 * -----
 * This function returns TRUE if the player is one or two rooms away
 * from the wumpus.
 */

static bool ISmellAWumpus(caveT cave)
{
    int i, r;

    for (i = 0; i < 3; i++) {
        r = cave->rooms[cave->player].connections[i];
        if (cave->wumpus == r) return (TRUE);
        if (AdjacentToWumpus(cave, r)) return (TRUE);
    }
    return (FALSE);
}

/*
 * Function: PrintStandings
 * Usage: PrintStandings(cave);
 * -----
 * This function prints the final won/lost standings.
 */

static void PrintStandings(caveT cave)
{
    printf("Your standings in this session are:\n");
    printf("  Player %3d\n", cave->playerWins);
    printf("  Wumpus %3d\n", cave->playerLosses);
}
```

```
/*
 * Function: GetYesOrNo
 * Usage: if (GetYesOrNo(prompt)) . . .
 * -----
 * This function asks the user the question indicated by prompt and
 * waits for a yes/no response. If the user answers "yes" (or "y")
 * or "no" ("n"), the program returns TRUE or FALSE accordingly. If
 * the user gives any other response, the program asks the question
 * again.
 */

static bool GetYesOrNo(string prompt)
{
    string answer;

    while (TRUE) {
        printf("%s", prompt);
        answer = ConvertToLowerCase(GetLine());
        if (StringEqual(answer, "yes")) return (TRUE);
        if (StringEqual(answer, "no")) return (FALSE);
        if (StringEqual(answer, "y")) return (TRUE);
        if (StringEqual(answer, "n")) return (FALSE);
        printf("Please answer yes or no.\n");
    }
}
```

```
/*
 * File: cave.h
 * -----
 * This file is the interface to the cave module, which exports
 * the data structures for the cave and the functions necessary
 * to initialize the cave structure.
 */

#ifndef _cave_h
#define _cave_h

#include "genlib.h"

/*
 * Constants
 * -----
 * NRooms    -- Number of rooms in the cave
 * NBats     -- Number of rooms containing bats
 * NPits     -- Number of rooms containing pits
 * NArrows   -- Number of arrows in player's quiver
 * MaxPath   -- Number of rooms to which the arrow can fly
 */

#define NRooms    20
#define NBats     3
#define NPits     3
#define NArrows   5
#define MaxPath   5

/*
 * Type: RoomT
 * -----
 * This type holds the data for a single room.  The only important
 * pieces of information are the room numbers of the three
 * connected rooms and flags indicating whether the room contains
 * bats or a pit.
 */

typedef struct {
    int connections[3];
    bool bat;
    bool pit;
} roomT;
```

```

/*
 * Type: CaveT
 * -----
 * This type holds the data for the entire cave. The individual
 * fields are used as follows:
 *
 *   rooms      Array of room structures (indices start at 1)
 *   player     Current location of the player
 *   wumpus     Current location of the wumpus
 *   arrows     Number of arrows remaining
 *   done       Flag used to determine when a game is finished
 *   playerWins Number of wins for the player
 *   playerLosses Number of losses
 */

typedef struct {
    roomT rooms[NRooms+1];
    int player;
    int wumpus;
    int arrows;
    bool done;
    int playerWins;
    int playerLosses;
} *caveT;

/*
 * Function: NewCave
 * Usage: cave = NewCave(cave);
 * -----
 * This function creates and returns a cave data structure. The
 * won/lost record is initialized to 0, but the cave topology
 * itself must be initialized by calling SetupCave.
 */

caveT NewCave(void);

/*
 * Function: SetupCave
 * Usage: SetupCave(cave);
 * -----
 * This function initializes the cave topology and populates the
 * cave with the various hazards. The function also initializes
 * the per-game data for the player (initial room, number of arrows).
 */

void SetupCave(caveT cave);

#endif

```



```
/*
 * File: cave.c
 * -----
 * This file implements the cave.h interface.
 */

#include <stdio.h>
#include "genlib.h"
#include "strlib.h"
#include "random.h"
#include "cave.h"

/* Private function prototypes */

static void CreateConnections(caveT cave);
static void PopulateCave(caveT cave);
static void Shuffle(int array[], int n);
static void SwapIntegerElements(int array[], int p1, int p2);

/* Exported entries */

/*
 * Function: NewCave
 * -----
 * This function allocates storage for a new cave structure
 * and initializes the won/lost record.
 */

caveT NewCave(void)
{
    caveT cave;

    cave = New(caveT);
    cave->playerWins = cave->playerLosses = 0;
    return (cave);
}

/*
 * Function: SetupCave
 * -----
 * This function initializes the cave topology. Most of the
 * work is done by subsidiary functions.
 */

void SetupCave(caveT cave)
{
    CreateConnections(cave);
    PopulateCave(cave);
    cave->arrows = NArrows;
}
```

```

/* Private functions */

/*
 * Function: CreateConnections
 * Usage: CreateConnections(cave);
 * -----
 * This function reads in the room connections from one of the
 * topology data files.
 */

static void CreateConnections(caveT cave)
{
    FILE *infile;
    string filename;
    int topology, i, r, c0, c1, c2;

    topology = RandomInteger(1, 9);
    filename = Concat("topology-",
                     Concat(IntegerToString(topology), ".dat"));
    infile = fopen(filename, "r");
    if (infile == NULL) {
        Error("The topology data files must be in the project folder");
    }
    for (i = 1; i <= NRooms; i++) {
        if (fscanf(infile, "%d: %d %d %d", &r, &c0, &c1, &c2) != 4 || i != r) {
            Error("Illegal connection format in topology file %s", filename);
        }
        cave->rooms[r].connections[0] = c0;
        cave->rooms[r].connections[1] = c1;
        cave->rooms[r].connections[2] = c2;
    }
}

/*
 * Function: PopulateCave
 * Usage: PopulateCave(cave);
 * -----
 * This function assigns the bats, pits, wumpus, and player to different
 * rooms. The strategy used to do so is to shuffle a list of room
 * numbers and then assign hazards to the rooms in that shuffled order.
 */

static void PopulateCave(caveT cave)
{
    int i, r, vp;
    int array[NRooms];

    for (i = 0; i < NRooms; i++) {
        r = array[i] = i + 1;
        cave->rooms[r].bat = cave->rooms[r].pit = FALSE;
    }
    Shuffle(array, NRooms);
    vp = 0;
    cave->player = array[vp++];
    cave->wumpus = array[vp++];
    for (i = 0; i < NBats; i++) {
        cave->rooms[array[vp++]].bat = TRUE;
    }
    for (i = 0; i < NPits; i++) {
        cave->rooms[array[vp++]].pit = TRUE;
    }
}

```

```
/*
 * Function: Shuffle
 * Usage: Shuffle(array, n);
 * -----
 * This function randomly shuffles the first n elements in the
 * specified integer array.
 */

static void Shuffle(int array[], int n)
{
    int lh, rh;

    for (lh = 0; lh < n; lh++) {
        rh = RandomInteger(lh, n-1);
        SwapIntegerElements(array, lh, rh);
    }
}

/*
 * Function: SwapIntegerElements
 * Usage: SwapIntegerElements(array, p1, p2);
 * -----
 * This function swaps the elements in array at index
 * positions p1 and p2.
 */

static void SwapIntegerElements(int array[], int p1, int p2)
{
    int tmp;

    tmp = array[p1];
    array[p1] = array[p2];
    array[p2] = tmp;
}
```

## Section 6

### Examinations

This section of the Instructor's Manual contains sample examinations used in CS106A, Stanford's equivalent of the CS1 course defined in Curriculum '78. Because Stanford is on the quarter system, there is only time for one midterm exam; on a semester calendar, two midterms might be preferable. The sample exams included in this section reflect the Stanford calendar and therefore consist of a single midterm administered in the fifth week of a ten-week quarter and a final, given during the regular examination period at the end of the term.

In addition to the actual examinations, this section also includes a "practice exam" for both the midterm and the final. These practice exams, handed out in class about a week before the real exam, are intended to serve as a detailed study guide. The real exam and practice exam have the same number of questions. Moreover, each question has the same title, style, and structure on each of the two exams. The idea is that if a student can finish the practice exam in the allotted time, that same student should then be able to complete the real exam as well. I have used the practice exam approach for many years—beginning long before arriving at Stanford—and I am convinced that it makes it possible to give harder exams that nonetheless result in significantly less student stress.

In CS106A, all exams are open-book. The advantage of using open-book exams is that you can focus the exams on the skills required for programming, as opposed to memorization of concepts or syntactic details. Forcing students to memorize everything they might need to know places undue emphasis on those details. After all, when students leave the university and take a programming job, they will be able to look things up from time to time. Open-book examinations—particularly when students are given a similar practice exam in advance—also tend to reduce the level of student stress.

The down side of open-book, programming-intensive exams is that they are harder to grade. At Stanford, each introductory course has a large staff of undergraduate teaching assistants who are responsible for most of the grading and make the overall task much more manageable. Stanford's approach to providing the teaching assistance required for these courses is discussed in the paper "Using undergraduates as teaching assistants in introductory programming courses: An update on the Stanford experience," which is included in Section 8.

## Practice Midterm Examination

---

This handout is intended to give you practice solving problems comparable in format and difficulty to those that will appear on the midterm examination next week. A solution set for this practice examination will be handed out in section. The sections this week will give you an opportunity to go over the solutions to the practice exam and to ask general questions about the course material.

### Problem 1—Karel the Robot (10 points)

*As noted in Section 4 (Designing the Course Syllabus), Stanford uses Richard Pattis's Karel the Robot system as a preprogramming exercise. This material is covered in the first question on the midterm and would only be relevant to schools that adopted the same approach.*

### Problem 2—Simple C expressions and statements (10 points)

- 2a. What is the value of the following C expression? What is its data type?

`6 / 4 * 2.0 + (7 * 8 % 9) * 1.2`

- 2b. Assume that the function `Mystery` has been defined as follows:

```
int Mystery(int x)
{
    int i, j;

    j = 0;
    for (i = 1; i < x; i++) {
        if (x % i == 0) j += i;
    }
    return (j);
}
```

What is the value of `Mystery(36)`?

- 2c. What output is displayed by the following program:

```
main()
{
    int x, y;

    x = 10;
    y = 17;
    x += y;
    y = x - y;
    x -= y;
    printf("x = %d, y = %d\n", x, y);
}
```

**Problem 3—Writing C functions and interfaces (10 points)**

Write a function `Max3` that takes three arguments of type `double` and returns the largest of those values. For example, calling

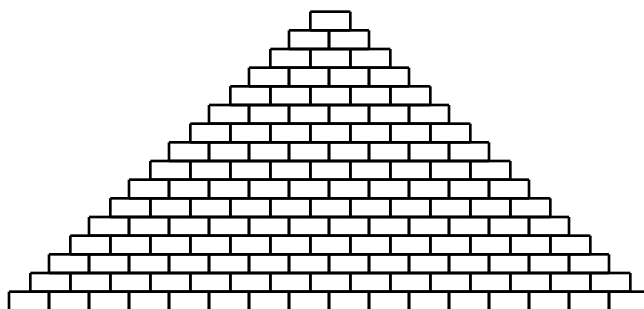
```
Max3(3.7, 11.4, 1.2)
```

should return 11.4.

In writing your answer, assume that your boss has asked you to make this function available to other users by defining an interface `max3.h` as well as the implementation `max3.c`. Write out the content of each of these two files. The interface should contain comments that describe the function to the client; the implementation need not include comments for this problem.

**Problem 4—Using the graphics library (15 points)**

Write a program that draws a pyramid consisting of bricks arranged in horizontal rows, so that the number of bricks in each row decreases by one as you move up the pyramid, as in this diagram:



Your implementation should use the constant `NBricksInBase` to specify the number of bricks in the bottom row and the constants `BrickWidth` and `BrickHeight` to specify the dimensions of each brick. To produce the figure in the diagram, these constants have the following values:

<code>BrickWidth</code>	.2
<code>BrickHeight</code>	.1
<code>NBricksInBase</code>	16

**Problem 5—Writing complete C programs (15 points)**

*Heads. . .*

*Heads. . .*

*Heads. . .*

*A weaker man might be moved to re-examine his faith, if in nothing else at least in the law of probability.*

— Tom Stoppard, *Rosencrantz and Guildenstern are Dead*, 1967.

Write a program that simulates flipping a coin repeatedly and continues until three *consecutive* "heads" are tossed. At that point, the program should display the total number of coin flips that were made. For example, a sample run of the program is

```
tails
heads
heads
tails
tails
heads
tails
heads
heads
heads
It took 10 flips to get 3 consecutive heads.
```

Write a complete program to solve this problem. You should design your program so that it will not give the same set of results each time you run it, but will instead use a different sequence each time. Remember that you can use functions that appear in the text either by providing the function name and chapter number or by including the appropriate interface.

## Solutions to Practice Midterm

---

**Problem 1—Karel the Robot (10 points)**

*Not relevant outside Stanford.*

**Problem 2—Simple C expressions and statements (10 points)**

**2a.**    Type:    `double`  
      Value:    4.4

**2b.**    This function computes the sum of the factors of `x`. Thus, `Mystery(36)` returns  
      `1 + 2 + 3 + 4 + 6 + 9 + 12 + 18`, or 55.

**2c.**    This program has the effect of exchanging the values of `x` and `y`. Thus, the output is

```
x = 17, y = 10
```



**Problem 3—Writing C functions and interfaces (10 points)**

```
/*
 * File: max3.h
 * -----
 * This file provides the interface for a library that
 * exports the function Max3, which returns the largest
 * of three arguments.
 */

#ifndef _max3_h
#define _max3_h

/*
 * Function: Max3
 * Usage: max = Max3(x, y, z);
 * -----
 * This function accepts three floating-point numbers and
 * returns the largest value.
 */

double Max3(double x, double y, double z);

#endif
```

```
/*
 * File: max3.c
 * -----
 * This file implements the max.h interface.
 */

#include <stdio.h>
#include "genlib.h"
#include "max3.h"

double Max3(double x, double y, double z)
{
    double max;

    max = x;
    if (y > max) max = y;
    if (z > max) max = z;
    return (max);
}
```

## Problem 4—Using the graphics library (15 points)

```
/*
 * File: pyramid.c
 * -----
 * This program draws a pyramid consisting of rows of
 * bricks stacked on top of each other, where the number
 * of bricks in each row is reduced by one for each row.
 */

#include <stdio.h>
#include "genlib.h"
#include "graphics.h"

/* Constants */

#define BrickWidth    .2
#define BrickHeight   .1
#define NBricksInBase 16

/* Function prototypes */

void DrawBox(double x, double y, double width, double height);
void DrawGrid(double x, double y, double width, double height,
               int columns, int rows);

/* Main program */

main()
{
    double cx, cy, x, y;
    int nb;

    InitGraphics();
    cx = GetWindowWidth() / 2;
    cy = GetWindowHeight() / 2;
    x = cx - NBricksInBase * BrickWidth / 2;
    y = cy - NBricksInBase * BrickHeight / 2;
    for (nb = NBricksInBase; nb > 0; nb--) {
        DrawGrid(x, y, BrickWidth, BrickHeight, nb, 1);
        x += BrickWidth / 2;
        y += BrickHeight;
    }
}

/* DrawBox and DrawGrid are as defined in Chapter 7 */
```

## Problem 5—Writing complete C programs (15 points)

```
/*
 * File: heads.c
 * -----
 * This program performs a random experiment consisting
 * of flipping a coin repeatedly until ConsecutiveHeads
 * heads appear in a row.
 */

#include <stdio.h>
#include "genlib.h"
#include "random.h"

/*
 * Constants
 * -----
 * ConsecutiveHeads -- number of consecutive heads desired
 */

#define ConsecutiveHeads 3

/* Main program */

main()
{
    int nFlips, nHeads;

    Randomize();
    nFlips = 0;
    nHeads = 0;
    while (nHeads < ConsecutiveHeads) {
        nFlips++;
        if (RandomChance(0.5)) {
            printf("Heads\n");
            nHeads++;
        } else {
            printf("Tails\n");
            nHeads = 0;
        }
    }
    printf("It took %d flips to get %d consecutive heads.\n",
          nFlips, ConsecutiveHeads);
}
```

## Midterm Examination

---

### General instructions

Answer each of the questions below. Write your answers directly on the examination paper, including any work that you wish to be considered for partial credit.

Each question is marked with the number of points assigned to that problem. The total number of points is 60. We intend the number of points to be roughly comparable to the number of minutes you should spend on that problem. This leaves you with half an hour to check your work or recover from false starts.

For all questions, you may include functions or definitions that have been developed in the course, either by writing the `#include` line for the appropriate interface (if there is one) or by giving the name of the function and the handout or chapter number in which a particular definition appears. For example, if you write

```
#include "graphics.h"
```

in your answer, you can then use any of the functions implemented in the graphics library.

Unless otherwise indicated as part of the instructions for a specific problem, comments will not be required on the exam. Uncommented code that gets the job done will be sufficient for full credit on the problem. On the other hand, comments may help you to get partial credit on a problem if they help us determine what you were trying to do.

The examination is open-book, and you may make use of any texts, handouts, or course notes. You may not, however, use a computer of any kind.

---

### THE STANFORD UNIVERSITY HONOR CODE

- A. The Honor Code is an undertaking of the students, individually and collectively:
- (1) that they will not give or receive aid in examinations; that they will not give or receive unpermitted aid in class work, in the preparation of reports, or in any other work that is to be used by the instructor as the basis of grading;
  - (2) that they will do their share and take an active part in seeing to it that others as well as themselves uphold the spirit and letter of the Honor Code.
- B. The faculty on its part manifests its confidence in the honor of its students by refraining from proctoring examinations and from taking unusual and unreasonable precautions to prevent the forms of dishonesty mentioned above. The faculty will also avoid, as far as practicable, academic procedures that create temptations to violate the Honor Code.
- C. While the faculty alone has the right and obligation to set academic requirements, the students and faculty will work together to establish optimal conditions for honorable academic work.

I acknowledge and accept the Honor Code.

(signed) \_\_\_\_\_

**Problem 1—Karel the Robot (10 points)**

*Not relevant outside Stanford.*

**Problem 2—Simple C expressions and statements (10 points)**

- 2a. What is the value of the following C expression? What is its type?

`10 * (9 + (8 * 7 - (6 + 5) % 4 * 3) * 2) + 1`

- 2b. Assume that the function `Enigma` has been defined as follows:

```
int Enigma(int n)
{
    int m;

    while (n >= 10) {
        m = 0;
        while (n > 0) {
            m += n % 10;
            n /= 10;
        }
        n = m;
    }
    return (n);
}
```

What is the value of `Enigma(1993)`?

- 2c. What output is displayed by the following program:

```
main()
{
    double x, y;
    int z;

    x = 16.5;
    y = 2.5;
    z = 0;
    while (x > y) {
        x -= y;
        z++;
    }
    printf("%d, %g\n", z, x);
}
```

**Problem 3—Writing C functions and interfaces (10 points)**

Write a function `IsSquare` that takes an integer  $n$  and returns `TRUE` if  $n$  is a perfect square, and `FALSE` otherwise. For example, calling

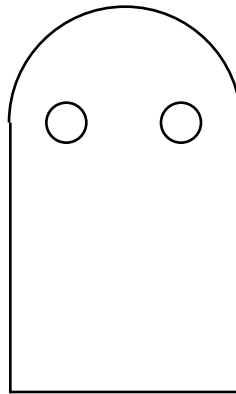
```
IsSquare(36)
```

should return `TRUE`.

In writing your answer, assume that your boss has asked you to make this function available to other users by defining an interface `issquare.h` as well as the implementation `issquare.c`. Write out the content of each of these two files. The interface should contain comments that describe the function to the client; the implementation need not include comments for this problem.

**Problem 4—Using the graphics library (15 points)**

Write a program that draws a picture of the Halloween ghost shown below:



The ghost should be centered on the screen, and should be based on the following constants:

<code>GhostWidth</code>	The width of the ghost, in inches
<code>GhostHeight</code>	The height of the ghost (from the base to the top of the head)
<code>EyeRadius</code>	The radius of each eye

The  $x$ -coordinates of the centers for the eyes should be  $1/4$  and  $3/4$  of the way across the ghost, and the  $y$ -coordinates should lie on the horizontal line that forms the base of the semicircle used for the head. Note that the radius of that semicircle is `GhostWidth / 2`.

In the example shown, the constants have the following values:

```
#define GhostHeight 2.0
#define GhostWidth  1.2
#define EyeRadius   0.1
```

In writing your program, make sure that changing these constants changes the shape of the picture accordingly.

**Problem 5—Writing complete C programs (15 points)**

Write a program to simulate a trick-or-treat outing as follows. For each of 10 “houses,” your program should display a message of the form

At house #\_\_, I got \_\_\_\_\_.

The first blank space should be filled in with the house number and the second one with one of the following five possible strings, chosen with equal probability:

candy corn  
a Tootsie Roll  
M&Ms  
an apple  
a rock

A sample run of this program might be

```
At house #1, I got M&Ms.  
At house #2, I got candy corn.  
At house #3, I got a Tootsie Roll.  
At house #4, I got M&Ms.  
At house #5, I got a rock.  
At house #6, I got candy corn.  
At house #7, I got an apple.  
At house #8, I got a Tootsie Roll.  
At house #9, I got M&Ms.  
At house #10, I got candy corn.
```

Write a complete program to solve this problem. You should design your program so that it will not give the same set of results each time it you run it, but will instead use a different sequence each time. Remember that you can use functions that appear in the text either by providing the function name and chapter number or by including the appropriate interface.

## Solutions to Midterm Exam

---

**Problem 1—Karel the Robot (10 points)**

*Not relevant outside Stanford.*

**Problem 2—Simple C expressions and statements (10 points)**

**2a.**    Type:    `int`  
      Value:    1031

**2b.**    This function performs an arithmetic operation called *casting out nines*, which some of you may have learned in elementary school as a way of checking answers in an arithmetic problem. The function begins by checking to see if there is more than one digit in the number. If so, it adds the digits and stores the result back in `n`. If the result is still greater than 10, it repeats the process. When `n` becomes less than 10, the function returns that value, which happens to be the remainder of the original number when divided by 9, except that if the original value of `n` is evenly divisible by 9, the result is 9 rather than 0.

Value:    4

**2c.**    This program has the effect of computing an integer quotient and a floating-point remainder. Thus, the result is

<code>6, 1.5</code>
---------------------



**Problem 3—Writing C functions and interfaces (10 points)**

```
/*
 * File: issquare.h
 * -----
 * This file provides the interface for a library that
 * exports the function IsSquare.
 */

#ifndef _issquare_h
#define _issquare_h

#include "genlib.h"          /* Required for bool */

/*
 * Function: IsSquare
 * Usage: if (IsSquare(n)) . . .
 * -----
 * This function returns TRUE if n is a perfect square.
 */

bool IsSquare(int n);

#endif
```

```
/*
 * File: issquare.c
 * -----
 * This file implements the issquare.h interface.
 */

#include <stdio.h>
#include <math.h>
#include "genlib.h"
#include "issquare.h"

bool IsSquare(int n)
{
    int root;

    root = sqrt(n) + 0.5;
    return (n == root * root);
}
```

## Problem 4—Using the graphics library (15 points)

```
/* File: ghost.c -- This program draws a Halloween ghost. */

#include <stdio.h>
#include "genlib.h"
#include "graphics.h"

/* Constants */

#define GhostHeight 2.0
#define GhostWidth  1.2
#define EyeRadius   0.1

/* Function prototypes */

void DrawGhost(double x, double y);
void DrawCenteredCircle(double x, double y, double r);

/* Main program */

main()
{
    double cx, cy;

    InitGraphics();
    cx = GetWindowWidth() / 2;
    cy = GetWindowHeight() / 2;
    DrawGhost(cx - GhostWidth / 2, cy - GhostHeight / 2);
}

/*
 * Function: DrawGhost
 * Usage: DrawGhost(x, y);
 * -----
 * This function draws a ghost whose lower-left corner is (x, y).
 */

void DrawGhost(double x, double y)
{
    double eyeHeight;

    eyeHeight = GhostHeight - GhostWidth / 2;
    MovePen(x, y);
    DrawLine(0, eyeHeight);
    DrawArc(GhostWidth / 2, 180, -180);
    DrawLine(0, -eyeHeight);
    DrawLine(-GhostWidth, 0);
    DrawCenteredCircle(x + 0.25 * GhostWidth, y + eyeHeight,
                       EyeRadius);
    DrawCenteredCircle(x + 0.75 * GhostWidth, y + eyeHeight,
                       EyeRadius);
}

/* DrawCenteredCircle is defined in Chapter 7 */
```

**Problem 5—Writing complete C programs (15 points)**

```
/*
 * File: treats.c
 * -----
 * This program simulates a trick-or-treat outing.
 */

#include <stdio.h>
#include "genlib.h"
#include "random.h"

/*
 * Constants
 * -----
 * NHouses -- number of houses
 */

#define NHouses 10

/* Function prototypes */

string RandomTreat(void);

/* Main program */

main()
{
    int i;

    Randomize();
    for (i = 1; i <= NHouses; i++) {
        printf("At house #%d, I got %s.\n", i, RandomTreat());
    }
}

string RandomTreat(void)
{
    switch (RandomInteger(1, 5)) {
        case 1: return ("candy corn");
        case 2: return ("a Tootsie Roll");
        case 3: return ("M&Ms");
        case 4: return ("an apple");
        case 5: return ("a rock");
    }
}
```

## Practice Final Examination

---

This handout is intended to give you practice solving problems comparable in format and difficulty to those that will appear on the final examination next week. A solution set for this practice examination will be handed out in Wednesday's class. On Friday, we will review the course and answer any questions you have about the solutions.

### Problem 1—Short answer (15 points)

- 1a. Some programs that work with points in three-dimensional space might represent those points as arrays of length three, so that a variable representing a point would be declared as

```
double pt[3];
```

Alternatively, you might use a record to define a point as follows:

```
typedef struct {
    double x, y, z;
} *point;

point pt;
```

What advantages can you suggest for each of these representations in C?

- 1b. Suppose that you have just come across the following comment and function prototype in an interface:

```
/*
 * Function: IsOrdered
 * Usage: if (IsOrdered(array) == TRUE) ...
 * -----
 * Using a for loop, this function compares
 * each element in the array to the following
 * element in that array; if the two are out
 * of sequence, the function returns FALSE.
 */

bool IsOrdered(int array[], int size);
```

The corresponding implementation (so that you can see what the function does) is

```
bool IsOrdered(int array[], int size)
{
    int i;

    for (i = 1; i < size; i++) {
        if (array[i-1] > array[i]) return (FALSE);
    }
    return (TRUE);
}
```

Identify at least four problems in the comments section of the interface. Rewrite those comments to eliminate the problems.

1c. Consider the following procedure definition:<sup>1</sup>

```
void f(int n)
{
    while (TRUE) {
        printf("%d\n", n);
        if (n == 1) break;
        if (n % 2 == 0) {
            n /= 2;
        } else {
            n = 3 * n + 1;
        }
    }
}
```

Trace through this procedure carefully and show what values are displayed when it is called with

**f(3)**

---

<sup>1</sup> As a side note for those who might go on into computer science theory, it is currently an open problem whether this procedure returns for all positive values of  $n$  (it might go into an infinite loop for some values of the argument). Proving that it always returns could make you famous someday.

**Problem 2—Simple C programming (15 points)**

Despite Einstein's metaphysical objections, the current models of physics, and particularly of quantum theory, are based on the idea that nature involves random processes. A radioactive atom, for example, does not decay for any specific reason that we mortals understand. Instead, that atom has a random probability of decaying within a period of time. Sometimes it does, sometimes it doesn't, and there is no way to know for sure.

Because physicists consider radioactive decay a random process, it is not surprising that random numbers can be used to simulate that process. Suppose you start with a collection of atoms, each of which has a certain probability of decaying in any unit of time. You can then approximate the decay process by taking each atom in turn and deciding randomly whether it decays.

Write a program that simulates the decay of a sample containing 10,000 atoms of radioactive material, where each atom has a 50 percent chance of decaying in a year. Your program should continue to run until the last atom has finally decayed.

The output of your program should be a table showing the year and the number of atoms remaining, such as the table in this sample run:

Year	Atoms left
0	10000
1	4969
2	2464
3	1207
4	627
5	311
6	166
7	89
8	40
9	21
10	8
11	4
12	1
13	0

As the numbers indicate, roughly half the atoms in the sample decay each year. In physics, the conventional way to express this observation is to say that the sample has a *half-life* of one year.

**Problem 3—Strings (15 points)**

The `strcmp` function from the `string.h` library operates by comparing character codes for each of the characters in the string. One sometimes unfortunate side-effect of this strategy is that character strings which differ only in the case of letters are considered different. For example, the string "CAT" is less than the string "cat" because the ASCII code for 'C' is less than the ASCII code for 'c'.

If you are writing a program that does indexing or alphabetization, it is often useful to be able to compare two strings without regard to the case of letters. If two strings contain completely different characters or if they are of different lengths, they will of course be considered different. If two strings match except for the fact that there is an uppercase letter in one string and the corresponding lowercase letter in the other, these strings should be considered the same.

Write a predicate function `EqualIgnoringCase` that takes two strings and returns `TRUE` if the strings contain the same characters if the case of letters is ignored. Thus,

```
EqualIgnoringCase("abc", "abc")           = TRUE
EqualIgnoringCase("aBc", "AbC")           = TRUE
EqualIgnoringCase("aBc", "AbCde")         = FALSE
EqualIgnoringCase("abc", "xyz")           = FALSE
```

**Problem 4—Arrays (15 points)**

When looking at a distribution of data values, there are three common statistical measures that capture the intuitive notion of an “average” element:

- The *mean*, which corresponds to the traditional average: the sum of all the data values divided by the number of elements
- The *median*, which is the value you get if you sort the distribution and then take the value at the middle of the sorted array
- The *mode*, which is simply the value that occurs most often

For example, in the array

43	65	72	75	79	82	82	84	84	84	86	90	94
0	1	2	3	4	5	6	7	8	9	10	11	12

the mean is

$$\frac{43 + 65 + 72 + 75 + 79 + 82 + 82 + 84 + 84 + 84 + 86 + 90 + 94}{13}$$

or 72, the median is the value in the midpoint element (`array[ 6 ]` or 82), and the mode is the value 84, since it occurs more often than any other value.

Write a C function `Mode` with the following prototype:

```
int Mode(int array[], int size)
```

The function takes an integer array and the actual number of elements contained, and returns the mode of the array. If there is more than one mode (which happens, for example, if two or more different values each occur more frequently than other values but themselves occur the same number of times), your program may return any of those values as the mode. Your program may *not* assume that the array is initially sorted, but you may sort it as part of the implementation of `Mode` by calling the `SortIntegerArray` function.

**Problem 5—CSim (10 points)**

*As noted in Section 4 (Designing the Course Syllabus), Stanford introduces a simple simulated computer system to give students a better understanding of machine architecture. The final exam usually contains a 10-point question on this material.*

**Problem 6—Data structure design (20 points)**

You have been assigned the task of computerizing the card catalog system for a library. In a prototype version of your library database, you want to store the following information for each of 1000 books:

- the title
- a list of up to five authors
- the Library of Congress catalog number
- up to five subject headings
- the publisher
- the year of publication
- whether the book is circulating or noncirculating

- 6a. Design the data structures that would be necessary to keep all the information for this prototype library database. Given your definition, it should be possible to write the declaration

```
libraryDB libdata;
```

and have the variable **libdata** contain all the information you would need to keep track of up to 1000 books. Keep in mind that the actual number of books will usually be less than this upper bound.

- 6b. Write a procedure **SearchBySubject** that takes as parameters the library database and a subject string. For each book in the library for which that subject string is listed as one of its subject headings, **SearchBySubject** should display the title, the name of the first author, and the Library of Congress catalog number of that book. You may assume that the database has already been initialized.



**Problem 7—Putting it all together (30 points)**

You have been hired as a programmer for a bank that has recently had a sharp increase in the number of customers who wish to exchange currency from one country for that of another. Heretofore, the tellers at the international desk have looked up the current international exchange rates and used a desk calculator to compute the correct amount of the new currency. The bank would like to automate this process and has assigned the task to you.

Assume that the bank will receive a data file each day containing the current exchange rates. The file name is **exchange-rates.dat**, and you may assume that the file is accessible in the directory of your project. The file itself contains lines of the form

DOLLAR	1.00
YEN	0.0078
FRANC	0.20
MARK	0.68
POUND	1.96

where each line consists of the name of some currency (always in upper case), at least one space, and the dollar equivalent of one unit of that currency. Thus, the sample file above tells us that the British Pound is worth \$1.96 and that the German Mark is worth 68 cents. (Note that the file includes a line for **DOLLAR** for which the exchange rate is always 1.00. The presence of this line means that U.S. dollars need not be treated as a special case.)

Write a program that performs the following steps:

1. Reads in the **exchange-rates.dat** data file into a suitable internal data structure.
2. Asks the user to enter two currency names: that of the old currency being converted and the new currency being returned.
3. Asks for a value in the original currency.
4. Displays the resulting value in the second currency. The easiest way to compute this value is probably to convert the original currency to dollars and then convert the dollars to the target currency.

A sample run of your program might be

```
Convert from: MARK
Into: YEN
How many units of type MARK? 200
200 MARK = 17435 YEN
```

## Solutions to Practice Final

---

### Problem 1—Short answer (15 points)

- 1a. The principal advantage of using the pointer-to-a-record definition is that it provides an encapsulation of the real world entity as a brand new type called `point`, which is distinguishable from all other types. If points are declared as arrays of three values of type `double`, the program offers no help to the reader as to the usage of that array. It looks just like any other array, and there is no name associated with the type that immediately identifies its usage. Similarly, the `point` type has components named `x`, `y`, and `z`—names that correspond to the traditional mathematical usage better than, for example, `pt[0]`. The main disadvantage of using the pointer-to-a-record model is that any points used in the program must be allocated using `New`.

One advantage of the array model is therefore that no separate allocation is required; declaring the array reserves the memory at the same time. Another potential advantage of the array model is that it allows the coordinates to be accessed using a `for` loop. For example, with the record model, you would have to name each of the components specifically if you wanted to perform some operation for each coordinate in the point. Using the array model, however, you could write a loop of the form

```
for (i = 0; i < 3; i++)
```

and then manipulate `pt[i]`.

- 1b. Here are several problems that need correcting:

- The “usage” section and the prototype do not match: the prototype indicates that an array size must be specified, but this parameter is missing from the “usage” line.
- The “usage” line has the redundant and inelegant construction `== TRUE`; as the textbook indicates, this error is a clear indication of a poor understanding of Boolean data.
- The description of the operation talks about the implementation. The client doesn’t want to know the details of *how* the implementation works, but instead needs to know *what* the implementation does.
- There is no indication as to the direction of the order. The client wants to know whether the function is looking for ascending or descending order.

Taking these concerns into account, you might rewrite the commentary as follows:

```
/*
 * Function: IsOrdered
 * Usage: if (IsOrdered(array, size)) . . .
 * -----
 * This function takes an array of integers and the
 * effective size of the array and returns TRUE if the
 * array is in ascending order, and FALSE otherwise.
 */
```

1c. Calling `f(3)` yields the following sample run:

```
3
10
5
16
8
4
2
1
```

**Problem 2—Simple C programming (15 points)**

```
/*
 * File: halflife.c
 * -----
 * This program simulates radioactive decay.
 */

#include <stdio.h>
#include "genlib.h"
#include "random.h"

/*
 * Constants
 * -----
 * InitialAtoms      -- Initial number of radioactive atoms
 * DecayProbability  -- Chance than an atom will decay in a year
 */

#define InitialAtoms      10000
#define DecayProbability  0.5

/* Main program */

main()
{
    int i, nAtoms, nDecay, year;

    printf("Year      Atoms left\n");
    printf("-----  ----- \n");
    nAtoms = InitialAtoms;
    for (year = 0; nAtoms > 0; year++) {
        printf("%3d      %6d\n", year, nAtoms);
        nDecay = 0;
        for (i = 0; i < nAtoms; i++) {
            if (RandomChance(DecayProbability)) nDecay++;
        }
        nAtoms -= nDecay;
    }
    printf("%3d      %6d\n", year, nAtoms);
}
```

**Problem 3—Strings (15 points)**

This problem has a perfectly correct, although somewhat inefficient, answer involving only one statement:

```
#include "strlib.h"

bool EqualIgnoringCase(string s1, string s2)
{
    return (StringEqual(ConvertToLowerCase(s1),
                        ConvertToLowerCase(s2)));
}
```

The more traditional approach is

```
#include <ctype.h>
#include "strlib.h"

bool EqualIgnoringCase(string s1, string s2)
{
    int i, len;

    len = StringLength(s1);
    if (len != StringLength(s2)) return (FALSE);
    for (i = 0; i < len; i++) {
        if (tolower(IthChar(s1, i)) != tolower(IthChar(s2, i)) {
            return (FALSE);
        }
    }
    return (TRUE);
}
```

**Problem 4—Arrays (15 points)**

As with many algorithmic problems, there are many ways to accomplish the task of finding a mode. Probably the most straightforward mechanism, but not the most efficient, is to ignore sorting entirely and just count the times each element appears, as follows:

```
/*
 * Function: Mode
 * Usage: mode = Mode(array, n);
 * -----
 * This implementation just goes through the array and
 * counts the number of times each element appears, keeping
 * track of the current maximum value.
 */

static int Mode(int array[], int n)
{
    int mode, max, count, i;

    max = 0;
    for (i = 0; i < n; i++) {
        count = CountIntegerElement(array[i], array, n);
        if (count > max) {
            max = count;
            mode = array[i];
        }
    }
    return (mode);
}

/*
 * Function: CountIntegerElement
 * Usage: n = CountIntegerElement(x, array, n);
 * -----
 * This function returns the number of times the element x
 * appears in the array.
 */

static int CountIntegerElement(int value, int array[], int n)
{
    int count, i;

    count = 0;
    for (i = 0; i < n; i++) {
        if (value == array[i]) count++;
    }
    return (count);
}
```

Alternatively, you could sort the array first and then just check the length of each run of consecutive elements. This is slightly more tricky, because the last run needs to be handled specially, as shown in the following program:

```
/*
 * Function: Mode
 * Usage: mode = Mode(array, n);
 * -----
 * This implementation sorts the array and then counts
 * the length of each run of consecutive equal elements.
 * Given the efficiency of the sort function from Chapter
 * 12, this approach is not actually more efficient than
 * the count-each-element strategy. With a better sorting
 * algorithm, such as the one presented in Chapter 17,
 * this strategy becomes considerably more efficient.
 * Note that an extra test must be made at the end of the
 * for loop to ensure that the last value is checked as a
 * possible mode.
 */

static int Mode(int array[], int n)
{
    int start, count, i, max, mode;

    SortIntegerArray(array, n);
    start = max = 0;
    count = 1;
    for (i = 1; i < n; i++) {
        if (array[start] == array[i]) {
            count++;
        } else {
            if (count > max) {
                mode = array[start];
                max = count;
            }
            start = i;
            count = 1;
        }
    }
    if (count > max) mode = array[start];
    return (mode);
}
```

**Problem 5—CSim (10 points)**

*Not relevant outside Stanford.*

## Problem 6—Data structure design (20 points)

6a.

```
/* Constants */

#define MaxBooks    1000
#define MaxAuthors   5
#define MaxSubjects  5

/*
 * Types
 * -----
 * bookT      -- The information about a particular book
 * libraryDB  -- The information about the library
 */

typedef struct {
    string title;
    string authors[MaxAuthors];
    int nAuthors;
    string subjects[MaxSubjects];
    int nSubjects;
    string catalogNumber;
    string publisher;
    int year;
    bool isCirculating;
} *bookT;

typedef struct {
    bookT collection[MaxBooks];
    int nBooks;
} *libraryDB;
```

6b.

```
void SearchBySubject(libraryDB libdata, string subject)
{
    int i;
    bookT book;

    for (i = 0; i < libdata->nBooks; i++) {
        book = libdata->collection[i];
        if (SubjectMatches(book, subject)) {
            printf("%s\n", book->title);
            printf("%s\n", book->authors[0]);
            printf("%s\n", book->catalogNumber);
            printf("\n");
        }
    }
}

bool SubjectMatches(bookT book, string subject)
{
    int i;

    for (i = 0; i < book->nSubjects; i++) {
        if (StringEqual(book->subjects[i], subject)) {
            return (TRUE);
        }
    }
    return (FALSE);
}
```

## Problem 7—Putting it all together (30 points)

```
/*
 * File: exchange.c
 * -----
 * This program uses a table of exchange rates stored as
 * a data file to convert from one currency unit to another.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"
#include "strlib.h"

/*
 * Constants
 * -----
 * MaxCurrencies    -- Maximum number of currencies
 * MaxCurrencyName  -- Maximum length of currency name
 * DatabaseFile     -- File name containing the currency database
 */

#define MaxCurrencies 100
#define MaxCurrencyName 25
#define DatabaseFile "exchange-rates.dat"

/*
 * Types
 * -----
 * currencyEntryT    -- Entry for single currency
 * exchangeDatabaseT -- Information for the exchange database
 */

typedef struct {
    string name;
    double dollars;
} currencyEntryT;

typedef struct {
    currencyEntryT info[MaxCurrencies];
    int nCurrencies;
} *exchangeDatabaseT;

/* Private function prototypes */

static exchangeDatabaseT ReadExchangeRates(void);
static int FindCurrency(exchangeDatabaseT db, string name);
```




```
/* Main program */

main()
{
    exchangeDatabaseT exchangeRates;
    string oldCurrency, newCurrency;
    int oldIndex, newIndex;
    double oldValue, dollars, newValue;

    exchangeRates = ReadExchangeRates();
    printf("Convert from: ");
    oldCurrency = GetLine();
    oldIndex = FindCurrency(exchangeRates, oldCurrency);
    if (oldIndex == -1) Error("No such currency");
    printf("Into: ");
    newCurrency = GetLine();
    newIndex = FindCurrency(exchangeRates, newCurrency);
    if (newIndex == -1) Error("No such currency");
    printf("How many units of type %s? ", oldCurrency);
    oldValue = GetReal();
    dollars = oldValue * exchangeRates->info[oldIndex].dollars;
    newValue = dollars / exchangeRates->info[newIndex].dollars;
    printf("%g %s = %g %s\n", oldValue, oldCurrency,
           newValue, newCurrency);
}

static exchangeDatabaseT ReadExchangeRates(void)
{
    exchangeDatabaseT db;
    FILE *infile;
    char namebuf[MaxCurrencyName], termch;
    int n, nscan;
    double rate;

    db = New(exchangeDatabaseT);
    infile = fopen(DatabaseFile, "r");
    if (infile == NULL) Error("No such file");
    n = 0;
    while (TRUE) {
        nscan = fscanf(infile, "%24s%lg%c",
                       namebuf, &rate, &termch);
        if (nscan == EOF) break;
        if (nscan != 3 || termch != '\n') {
            Error("Illegal format in database file");
        }
        if (n == MaxCurrencies) Error("Too many currencies");
        db->info[n].name = ConvertToLowerCase(namebuf);
        db->info[n].dollars = rate;
        n++;
    }
    fclose(infile);
    db->nCurrencies = n;
    return (db);
}
```



```
/*
 * Function: FindCurrency
 * Usage: index = FindCurrency(db, name);
 * -----
 * This function looks up the name of the currency in the
 * database and returns the index of that entry. If there
 * is no currency with the indicated name, FindCurrency
 * returns NULL.
 */

static int FindCurrency(exchangeDatabaseT db, string name)
{
    int i;

    name = ConvertToLowerCase(name);
    for (i = 0; i < db->nCurrencies; i++) {
        if (StringEqual(db->info[i].name, name)) return (i);
    }
    return (-1);
}
```

## Final Examination

---

### General instructions

Answer each of the questions below. Write your answers directly on the examination paper, including any work that you wish to be considered for partial credit.

Each question is marked with the number of points assigned to that problem. The total number of points is 120. We intend that the number of points be roughly comparable to the number of minutes you should spend on that problem. This leaves you with an hour to check your work or to recover from false starts.

For all questions, you may include functions or definitions that have been developed in the course, either by writing the `#include` line for the appropriate interface (if there is one) or by giving the name of the function and the handout or chapter number in which a particular definition appears. For example, if you write

```
#include "graphics.h"
```

in your answer, you can then use any of the functions implemented in the graphics library.

Unless otherwise indicated as part of the instructions for a specific problem, comments will not be required on the exam. Uncommented code that gets the job done will be sufficient for full credit on the problem. On the other hand, comments may help you to get partial credit on a problem if they help us determine what you were trying to do.

The examination is open-book, and you may make use of any texts, handouts, or course notes. You may not, however, use a computer of any kind.

**Problem 1—Short answer (15 points)**

- 1a. In a short paragraph, describe the advantages and disadvantages of using global variables. Why is it important that global variables are declared using the **static** keyword?
- 1b. Suppose that you have been hired by a company to take over development of a library package containing several useful extensions to the graphics library. Your predecessor on the project has completed the interface design and written a draft implementation. Unfortunately, the implementation is buggy, and your job is to fix it.

One of the functions in the new library package is **DrawRoundedBox**, which is in fact a solution to Exercise 5 on page 254. The **DrawRoundedBox** function has the following interface description:

```
/*
 * Function: DrawRoundedBox
 * Usage: DrawRoundedBox(x, y, width, height);
 * -----
 * This function is identical in operation to DrawBox except that
 * the corners of the box are replaced by quarter circles, which
 * gives the box a rounded appearance. The parameters x, y, width,
 * and height give the coordinates and dimensions of the complete
 * rectangular box; the rounded box will cover the same area
 * except at the rounded corners. The radius of the quarter
 * circle at the corner is given by the constant CornerRadius.
 * This value, however, is only used if the box is at least twice
 * that large in each dimension. If not, the circle radius is
 * chosen to be half of the smaller of the width and the height.
 */
```

The implementation, as written by your predecessor, looks like this:

```
void DrawRoundedBox(double x, double y, double width, double height)
{
    double r;

    r = CornerRadius;
    if (CornerRadius > width / 2) r = width / 2;
    if (CornerRadius > height / 2) r = height / 2;
    MovePen(x, y);
    DrawLine(width - 2 * r, 0);
    DrawArc(r, -90, 90);
    DrawLine(0, height - 2 * r);
    DrawArc(r, 0, 90);
    DrawLine(-width - 2 * r, 0);
    DrawArc(r, 90, 90);
    DrawLine(0, -height - 2 * r);
    DrawArc(r, 180, 90);
}
```

Work through the program by hand to identify the bugs. Rewrite the implementation so that it matches the specification given in the interface.

1c. Suppose that the integer array **array** has been declared and initialized as follows:

```
#define NElements 5

int list[NElements];

list[0] = 10;
list[1] = 20;
list[2] = 30;
list[3] = 40;
list[4] = 50;
```

This code sets up an array of five elements with the initial values shown in the following diagram:

list				
10	20	30	40	50

Given this array, what is the effect of calling the function

```
Mystery(list, NElements);
```

if **Mystery** is defined as

```
void Mystery(int array[], int n)
{
    int i, tmp;

    tmp = array[n - 1];
    for (i = 1; i < n; i++) {
        array[i] = array[i - 1];
    }
    array[0] = tmp;
}
```

Work through the function carefully and indicate your answer by filling in the boxes below to show the final contents of **list**.

list				

**Problem 2—Simple C programming (15 points)**

*Golf is a game in which you put a ball an inch-and-a-half in diameter on top of a ball 8,000 miles in diameter and try to hit just the smaller one.*

— Anonymous

In this problem, your task is to write a simple simulation of the game of golf. To simplify the problem, assume that the game is played only in a single dimension (i.e., the ball never leaves a line drawn from the tee through the hole) so that the only parameter you need to consider is the distance from the hole. Your simulation should include the following steps.

1. Pick a random distance for the hole from 250 to 500 yards.
2. For each turn, repeat each of the steps 2a–2d below until the distance to the hole is less than 25 yards.
  - 2a. Display the distance remaining to the hole.
  - 2b. Let the user select a club by typing a number between 1 and 9.
  - 2c. Assume that the average distance the ball goes is given by the formula

$$\text{average-distance} = 300 - \text{club-number} * 30$$

so that a #1 club goes an average distance of  $300 - 1 * 30$  (270 yards) and a #9 club goes an average distance of  $300 - 9 * 30$  (30 yards). Once you have computed the average distance, you should add in a random integer between -20 and +20, inclusive, just to make the game less repetitive.

- 2d. Subtract the distance computed in step 2c from the distance remaining to the hole. Note that it is possible to overshoot the hole; you should make sure your program always uses a positive value for the remaining distance to the hole.
3. When the remaining distance to the hole is less than 25 yards, tell the user that he or she has reached the green, display the number of strokes so far, and stop.

Write a program that simulates the golf game described above. The following is a sample run of the program:

```
Let's play golf!
You're on the tee
You're 384 yards from the pin
Which club? 1
Your shot went 250 yards
You're 134 yards from the pin
Which club? 6
Your shot went 109 yards
You're 25 yards from the pin
Which club? 9
Your shot went 21 yards
You're on the green after 3 strokes
```

**Problem 3—Strings (15 points)**

A Roman numeral is written as a string of letters, in which each letter stands for a value as shown in the following table:

I	1
V	5
X	10
L	50
C	100
D	500
M	1000

The value of a Roman numeral is given by adding up the values corresponding to each letter. Thus, the string

MDCCCCLXXXIII

corresponds to

$$1000 + 500 + 100 + 100 + 100 + 100 + 50 + 10 + 10 + 10 + 10 + 1 + 1 + 1$$

or 1993.

Write a function **RomanToDecimal** that takes a string as an argument and returns the equivalent integer by going through the string and adding up the values corresponding to each character. If an illegal character appears (i.e., a character that is not one of the uppercase letters M, D, C, L, X, V, or I), your implementation should call **Error** with an appropriate message.

Note that your function is required to handle Roman numerals only in their simple additive form. You should not try to make your function handle the form of Roman numeral in which a smaller value written in front of a larger one indicates subtraction. Thus, your program should interpret IX as 11 (the sum of the values) and not as 9.

**Problem 4—Arrays (15 points)**

As noted in Chapter 11, an *identity matrix* is a two-dimensional array of floating-point numbers whose elements are 1.0 along the diagonal and 0.0 everywhere else, as in

1.0	0.0	0.0
0.0	1.0	0.0
0.0	0.0	1.0

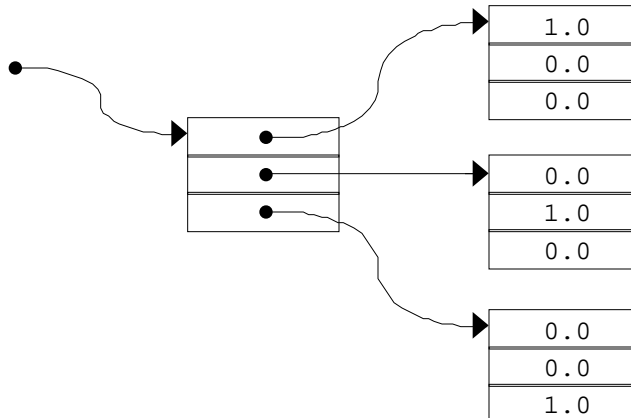
Although an identity matrix is always square (which means that it has the same number of rows and columns), it may be of any size. Thus, the preceding diagram shows the identity matrix of size 3, while the identity matrix of size 4 looks like this:

1.0	0.0	0.0	0.0
0.0	1.0	0.0	0.0
0.0	0.0	1.0	0.0
0.0	0.0	0.0	1.0

In this problem, your job is to write a function `IdentityMatrix(size)` that dynamically allocates, initializes, and returns an identity matrix of the correct size. The prototype for the function is

```
double **IdentityMatrix(int size);
```

which underscores the fact that the dynamically allocated matrix is actually an array of arrays, where each array is internally represented as a pointer. For example, the result of calling `IdentityMatrix(3)` has the following structure in memory:





**Problem 5—CSim (10 points)***Not relevant outside Stanford.***Problem 6—Data structure design (20 points)**

As our culture becomes more health-conscious, people are increasingly concerned about whether the food they eat supplies the necessary nutrition for the day. Today, most food packaging includes a table that shows how much of the recommended daily intake of various nutrients a single serving of that food supplies. For example, a typical table might look like this:

<b>Chocolate Frosted Sugar Bombs</b>	
Total Fat	5%
Cholesterol	0%
Carbohydrate	125%
Sodium	10%
Vitamin A	5%
Vitamin C	5%

You have been given the task of designing the data structures necessary to keep track of the nutritional information for a set of foods. Your database should be able to handle up to 100 different products and to store the nutritional information for each.

All foods have a product name; beyond that, however, the list of nutrients will vary significantly depending on the product. Thus, you cannot assume that all foods will have an entry for “Sodium”; some will and some won’t. The implication is that you will have to store the name of each nutrient explicitly as part of the structure. The number of nutrients per food will never exceed 25.

- 6a.** Design the data structures that would be necessary to maintain the product nutrition database. Your definition should include a type `nutritionDatabaseT` that is capable of representing all the necessary information within that structure.
- 6b.** Write a procedure `ListSufficientFoods(db,nutrient)` that takes as parameters the nutrition database and the name of a particular nutrient. Your procedure should then list every food in the database that supplies at least 100% of the recommended amount of the specified nutrient along with the percentage of that nutrient supplied, ignoring entirely any foods that don’t list that nutrient at all. Thus, if you called

```
ListSufficientFoods(db, "Carbohydrate");
```

on a database that included the sample food above, the output would include the line

```
Chocolate Frosted Sugar Bombs      125%
```

**Problem 7—Putting it all together (30 points)**

The year: 1984

The place: A parallel universe more like George Orwell's vision.

The newly revived and strengthened House Un-American Activities Commission Task Force on Subversive Literature has assigned you the task of developing a program that will scan novels, textbooks, and other printed matter for subversive content.

Write a program that performs the following steps:

1. Reads in a list of words from a data file called `censor.dat`. Each line of the `censor.dat` file consists of a word, at least one space, and a “subversion index” (always an integer) indicating the danger such a word represents to the public order—at least in the minds of those on the HUAC committee. For example, the file might contain the following lines:

```
communist 20
equality 4
feminism 12
liberal 9
pc 10
revolution 15
```

2. Reads in a line from the user.
3. Separates the input line into words and calculates the subversion index of each word. If a word does not appear in the data file, its subversion index is 0.
4. Displays the total subversion index for the line.

The following sample run illustrates the operation of the program:

```
Enter a line of text to be analyzed.
The Mac represents a revolution in PC design.
Total subversion index = 25
```

Before you start writing this program, think about the tools you already have. Like most programs, this one is much easier if you can make use of existing code. Remember that you can use any function or library in the text just by giving its name.

## Solutions to Final Examination

---

### Problem 1—Short answer (15 points)

- 1a. The principal advantage of using global variables is that they allow an abstraction to maintain internal state between calls. The main disadvantage of using them is that there is no way to control which functions can access and change their values. Declaring a global variable to be **static** provides some protection, because it limits the visibility of the global variable to a single program module.
- 1b. The bugs are that
- The initialization of **r** is incorrect. If **width** is less than **height** and both are less than **CornerRadius**, **r** will be set to the wrong value.
  - The first line is not inset from the corner.
  - The **DrawLine** commands that draw the top and left sides of the box need parentheses to ensure that the negative signs apply to the entire quantity. (As an alternative, the interior - signs can be replaced by + signs.)

The following code corrects the bugs:

```
void DrawRoundedBox(double x, double y, double width, double height)
{
    double r;

    r = CornerRadius;
    if (r > width / 2) r = width / 2;
    if (r > height / 2) r = height / 2;
    MovePen(x + r, y);
    DrawLine(width - 2 * r, 0);
    DrawArc(r, -90, 90);
    DrawLine(0, height - 2 * r);
    DrawArc(r, 0, 90);
    DrawLine(-(width - 2 * r), 0);
    DrawArc(r, 90, 90);
    DrawLine(0, -(height - 2 * r));
    DrawArc(r, 180, 90);
}
```

1c.

list

50	10	10	10	10
----	----	----	----	----

## Problem 2—Simple C programming (15 points)

```
/*
 * File: golf.c
 * -----
 * This program simulates a golf game.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"
#include "random.h"

/*
 * Constants
 * -----
 * MinDistance    Minimum distance from tee to hole
 * MaxDistance    Maximum distance from tee to hole
 * GreenSize      Radius of area considered to be the green
 * FudgeYards     Random error introduced into each shot
 */

#define MinDistance 250
#define MaxDistance 500
#define GreenSize   25
#define FudgeYards  20

/* Function prototypes */

int ShotDistance(int club);
int GetClub(void);

main()
{
    int yardsLeft, nYards, club, strokes;

    Randomize();
    printf("Let's play golf!\n");
    printf("You're on the tee\n");
    yardsLeft = RandomInteger(MinDistance, MaxDistance);
    strokes = 0;
    while (yardsLeft >= GreenSize) {
        printf("You're %d yards from the hole.\n", yardsLeft);
        club = GetClub();
        nYards = ShotDistance(club);
        printf("Your shot went %d yards\n", nYards);
        yardsLeft = abs(yardsLeft - nYards);
        strokes++;
    }
    printf("You're on the green after %d strokes\n", strokes);
}
```

```
/*
 * Function: ShotDistance
 * Usage: nYards = ShotDistance(club);
 * -----
 * This function calculates how far the ball goes, based on
 * the club number, as specified by the problem.
 */

int ShotDistance(int club)
{
    int yards;

    yards = 300 - club * 30;
    yards += RandomInteger(-FudgeYards, FudgeYards);
    return (yards);
}

/*
 * Function: GetClub
 * Usage: club = GetClub();
 * -----
 * This function asks the user for a club number and returns it.
 * If the value entered is not a legal club number, the user gets
 * another chance to enter it.
 */

int GetClub(void)
{
    int club;

    while (TRUE) {
        printf("Which club? ");
        club = GetInteger();
        if (club >= 1 && club <= 9) return (club);
        printf("Club number must be between 1 and 9\n");
    }
}
```

## Problem 3—Strings (15 points)

```
/*
 * Function: RomanToDecimal
 * Usage: decimal = RomanToDecimal(roman);
 * -----
 * This function converts a Roman numeral to its decimal equivalent.
 */

int RomanToDecimal(string roman)
{
    int i, total;

    total = 0;
    for (i = 0; i < StringLength(roman); i++) {
        total += RomanDigit(IthChar(roman, i));
    }
    return (total);
}

/*
 * Function: RomanDigit
 * Usage: n = RomanDigit(letter);
 * -----
 * This function converts a letter to its decimal equivalent.
 */

int RomanDigit(char letter)
{
    switch (letter) {
        case 'M': return (1000);
        case 'D': return (500);
        case 'C': return (100);
        case 'L': return (50);
        case 'X': return (10);
        case 'V': return (5);
        case 'I': return (1);
        default: Error("Illegal letter %c", letter);
    }
}
```

**Problem 4—Arrays (15 points)**

```
/*
 * Function: IdentityMatrix
 * Usage: matrix = IdentityMatrix(size);
 * -----
 * This function allocates and returns an identity matrix of the
 * specified size.
 */

double **IdentityMatrix(int size)
{
    double **matrix;
    int i, j;

    matrix = NewArray(size, double *);
    for (i = 0; i < size; i++) {
        matrix[i] = NewArray(size, double);
        for (j = 0; j < size; j++) {
            matrix[i][j] = (i == j) ? 1.0 : 0.0;
        }
    }
    return (matrix);
}
```

**Problem 5—CSim (10 points)**

*Not relevant outside Stanford.*

**Problem 6—Data structure design (20 points)**

6a.

```
/*
 * Constants
 * -----
 * MaxFoods      -- Maximum number of foods
 * MaxNutrients  -- Maximum number of nutrients per food
 */

#define MaxFoods      100
#define MaxNutrients  25

/*
 * Types
 * -----
 * nutrientT      -- Information about a particular nutrient
 * foodT          -- Information about a particular food
 * nutrientDatabaseT -- Information about the entire collection
 */

typedef struct {
    string nutrient;
    int percentage;           /* Could also be double */
} nutrientT;

typedef struct {
    string name;
    nutrientT nutrients[MaxNutrients];
    int nNutrients;
} *foodT;

typedef struct {
    foodT foods[MaxFoods];
    int nFoods;
} *nutrientDatabaseT;
```



6b.

```
/*
 * Function: ListSufficientFoods
 * Usage: ListSufficientFoods(db, nutrient);
 * -----
 * This function lists the names of all foods in the database
 * that supply at least 100 percent of the indicated nutrient
 * along with the percentage of that nutrient.
 */

void ListSufficientFoods(nutrientDatabaseT db, string nutrient)
{
    int i, percent;

    for (i = 0; i < db->nFoods; i++) {
        percent = GetNutrientPercentage(db->foods[i], nutrient);
        if (percent >= 100) {
            printf("%-35s%3d%\n", db->foods[i]->name, percent);
        }
    }
}

/*
 * Function: GetNutrientPercentage
 * Usage: percent = GetNutrientPercentage(food, nutrient);
 * -----
 * This function returns the percentage of a specified nutrient
 * for a given food product.  If the food product does not list
 * that nutrient, GetNutrientPercentage returns -1.
 */

int GetNutrientPercentage(foodT food, string nutrient)
{
    int i;

    for (i = 0; i < food->nNutrients; i++) {
        if (StringEqual(food->nutrients[i].nutrient, nutrient)) {
            return (food->nutrients[i].percentage);
        }
    }
    return (-1);
}
```

## Problem 7—Putting it all together (30 points)

```
/*
 * File: censor.c
 * -----
 * This program rates a line of text to see how subversive
 * it is. To do so, the program checks each word in the line
 * to see if it matches one of a set of forbidden words. The
 * offending words are stored in a file along with an integer
 * indicating the "subversion index" for each word. The total
 * index for the line is the sum of the subversion index for
 * each word.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"
#include "strlib.h"
#include "scanner.h"

/*
 * Constants
 * -----
 * MaxWords      -- Maximum number of words in file
 * MaxWordLength -- Maximum length of word
 * CensorFile    -- File containing the subversive words
 */

#define MaxWords      100
#define MaxWordLength 25
#define CensorFile    "censor.dat"

/*
 * Types
 * -----
 * wordEntryT -- Entry for a single word
 * dictionaryT -- The entire subversive word dictionary
 */

typedef struct {
    string word;
    int subversion;
} wordEntryT;

typedef struct {
    wordEntryT words[MaxWords];
    int nWords;
} *dictionaryT;

/* Private function prototypes */

static dictionaryT ReadDictionary(void);
static int SubversionIndex(dictionaryT dict, string word);
```

```
/* Main program */


main()
{
    dictionaryT dict;
    string line;
    int subversion;

    dict = ReadDictionary();
    printf("Enter a line of text to be analyzed.\n");
    line = GetLine();
    subversion = 0;
    InitScanner(line);
    while (!AtEndOfLine()) {
        subversion += SubversionIndex(dict, GetNextToken());
    }
    printf("Total subversion index = %d\n", subversion);
}

/*
 * Function: ReadDictionary
 * Usage: dict = ReadDictionary();
 * -----
 * This function reads the subversive word dictionary into
 * an internal data structure and returns that structure.
 */

static dictionaryT ReadDictionary(void)
{
    dictionaryT dict;
    FILE *infile;
    char wordbuf[MaxWordLength], termch;
    int n, nscan, subversion;

    dict = New(dictionaryT);
    infile = fopen(CensorFile, "r");
    if (infile == NULL) Error("No such file");
    n = 0;
    while (TRUE) {
        nscan = fscanf(infile, "%24s%d%c",
                       wordbuf, &subversion, &termch);
        if (nscan == EOF) break;
        if (nscan != 3 || termch != '\n') {
            Error("Illegal format in dictionary");
        }
        if (n == MaxWords) Error("Too many words");
        dict->words[n].word = ConvertToLowerCase(wordbuf);
        dict->words[n].subversion = subversion;
        n++;
    }
    fclose(infile);
    dict->nWords = n;
    return (dict);
}
```



```
/*
 * Function: SubversionIndex
 * Usage: subversion = SubversionIndex(dict, word);
 * -----
 * This function looks up the word in the censorship dictionary
 * and returns its subversion index if that word appears.  If the
 * word is not there (which is presumably true for punctuation
 * tokens as well), the function returns 0.
 */

static int SubversionIndex(dictionaryT dict, string word)
{
    int i;

    word = ConvertToLowerCase(word);
    for (i = 0; i < dict->nWords; i++) {
        if (StringEqual(dict->words[i].word, word)) {
            return (dict->words[i].subversion);
        }
    }
    return (0);
}
```

## Section 7

# Solutions to Exercises

This section of the Instructor's Manual contains the solutions to all the review questions and exercises in the text.

The exercises are useful as short programming problems. Particularly late in the term, the Stanford introductory course tends to assign larger programming problems than those in the text, so the students gain experience working with more substantial software projects than is customary in an introductory course. These additional assignments are described in more detail in Section 5 (Assignments).

# Solutions for Chapter 1

## Introduction

### Review questions

1. Babbage's Analytical Engine introduced the concept of programming to computing.
2. Augusta Ada Byron is generally recognized as the first programmer. The U.S. Department of Defense named the Ada programming language in her honor.
3. The heart of von Neumann architecture is the stored-programming concept, in which both data and programming instructions are stored in the same memory system.
4. Hardware is tangible and comprises the physical parts of a computer; software is intangible and consists of the programs the computer executes.
5. The abstract concept that forms the core of computer science is problem-solving.
6. For a solution technique to be an algorithm, it must be
  - a. *Clearly and unambiguously defined*
  - b. *Effective*, in the sense that its steps are executable
  - c. *Finite*, in the sense that it terminates after a bounded number of steps
7. *Algorithmic design* refers to the process of designing a solution strategy to fit a particular problem; *coding* refers to the generally simpler task of representing that solution strategy in a programming language.
8. A higher-level language is a programming language that is designed to be independent of the particular characteristics that differentiate computers and to work instead with general algorithmic concepts that can be implemented on any computer system. The higher-level language used in this text is called ANSI C.
9. Each type of computer has its own machine language, which is different from that used in other computers. The compiler acts as a translator from the higher-level language into the machine language used for a specific machine.
10. A source file contains the actual text of a program and is designed to be edited by people. An object file is created by the compiler and contains a machine-language representation of the program. Most programmers never work directly with object files.
11. A syntax error is a violation of the grammatical rules of the programming language. The compiler catches syntax errors when it translates your program. Bugs are errors of the logic of the program. Programs containing bugs are perfectly legal in the sense that the compiler can find no violation of the rules, but nonetheless they do not behave as the programmer intended.
12. False. Even the best programmers make mistakes.
13. False. Between 80 and 90 percent of the cost of a program comes from maintaining that program after it is put into practice.
14. The term *software maintenance* refers to development work on a program that continues after the program has been written and released. Although part of software maintenance involves fixing bugs that were not detected during initial testing, most maintenance consists of adapting a program to meet new requirements.
15. Programs are usually modified many times over their lifetimes. Programs that use good software engineering principles are much easier for other programmers to understand and are therefore easier to maintain.

## Solutions for Chapter 2

### Learning by Example

#### Review questions

1. The purpose of the comment at the beginning of each program is to convey information to the human reader of the program about what it does.
2. A programming library is a collection of tools written by other programmers that allow you to perform various useful operations without having to write the necessary code yourself.
3. To use the system mathematical library, your program must include the line

```
#include <math.h>
```

4. To use the graphics library, your program must include the line

```
#include "graphics.h"
```

The quotation marks are used here to indicate that this library is not part of the standard ANSI library collection.

5. Every complete C program must define a function called **main**.
6. The special character `\n` is called the *newline character* and indicates that the cursor should be moved to the beginning of the following line. When you want to request input from the user, it is usually best for the cursor to remain immediately after the prompt string so that it is clearer to the user what kind of input the program needs.
7. An argument is information that the caller of a particular function makes available to the function itself. Arguments increase the power of the function-calling mechanism because the same function can be applied to different data values.
8. The necessary declarations are

```
int voteCount1, voteCount2;  
double x, y, z;
```

9. The three phases of the programs in this chapter are: *input*, *computation*, *output*.
10. The **GetInteger** function makes it easier to read in an integer value from the user. In most cases, programs first display a prompt that tells the user what to enter and then call **GetInteger** to read that value, as illustrated by the following lines:

```
printf("1st number? ");  
n1 = GetInteger();
```

11. The sequences `%d` and `%g` are called *format codes* and are used as placeholders for values that **printf** will insert into those positions. The format code `%d` is used to display a decimal integer, and `%g` is used to display a floating-point value.
12. Reductionism is the philosophical theory that the best way to comprehend a large system is to understand in detail the parts that comprise it. By contrast, holism is the view that the whole is greater than the sum of the parts and must be understood as a separate entity unto itself. Programmers must learn to view their work from both perspectives. Getting the details right is critical to accomplishing the task. At the same time, setting those details aside and focusing on the big picture often makes it easier to manage the complexity involved in programming.
13. A data type is defined by a domain and a set of operations.
14. All programs throughout the remainder of this text include the line

```
#include "genlib.h"
```

15. a. legal, integer  
b. legal, integer  
c. illegal (contains an operator)  
d. legal, floating-point  
e. legal, integer  
f. legal, floating-point  
g. illegal (contains commas)  
h. legal, floating-point  
i. legal, integer  
j. legal, floating-point  
k. legal, floating-point  
l. illegal (contains the letter **x**)
16. a. **6.02252E+23**  
b. **2.997925E+10**  
c. **5.29167E-10**  
d. **3.1415926535E0**
17. a. legal  
b. legal  
c. legal  
d. illegal (contains %)  
e. illegal (**short** is reserved)  
f. legal  
g. illegal (contains a space)  
h. legal  
i. illegal (begins with a digit)  
j. illegal (contains a hyphen)  
k. legal  
l. legal
18. You can make no assumptions about the value of a variable until you explicitly give it a value in your program.
19. a. **int: 5**  
b. **int: 3**  
c. **double: 3.8**  
d. **double: 18.0**  
e. **int: 4**  
f. **int: 2**
20. The variable **k** would contain the value 3 after the first assignment and 2 after the second. When a floating-point value is assigned to an integer variable, its value is *truncated* by dropping any decimal fraction.
21. The unary minus operator is written before a single term, as in **-x**; the binary subtraction operator uses the same symbol but is written between two terms, as in **x - 3**.
22. a. 4  
b. 2  
c. 42  
d. 42
23. In C, conversion between numeric types can be indicated explicitly using a type cast, which is composed of a type name in parentheses, followed by the expression to be converted.

### Programming exercises

1. The **hello.c** program appears on page 23 of the text.



2. This program calculates the area of a triangle given its base and height. To make its purpose clearer, it needs to be rewritten to include comments, instructions to the user, and more easily recognized variable names.

```
/*
 * File: triangle.c
 * -----
 * This program calculates the area of a triangle using the
 * formula:
 *           base * height
 *   Area =  -----
 *           2
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
    double base, height, area;

    printf("This program computes the area of a triangle\n");
    printf("given its base and height.\n");
    printf("Enter triangle base: ");
    base = GetReal();
    printf("Enter triangle height: ");
    height = GetReal();
    area = (base * height) / 2;
    printf("Area of triangle = %g\n", area);
}
```

- 3.
- ```
/*
 * File: feettocm.c
 * -----
 * This program reads in a length given in feet and inches and
 * converts it to its metric equivalent in centimeters.
 */
```

```
#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
    double feet, inches, centimeters;

    printf("This program converts from feet and inches to centimeters.\n");
    printf("Number of feet? ");
    feet = GetReal();
    printf("Number of inches? ");
    inches = GetReal();
    centimeters = (feet * 12 + inches) * 2.54;
    printf("The corresponding length is %g cm.\n", centimeters);
}
```

4.

```
/*
 * File: interest.c
 * -----
 * This program calculates simple interest for one year.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
    double balance, rate, interest;

    printf("Interest calculation program.\n");
    printf("Starting balance? ");
    balance = GetReal();
    printf("Annual interest rate percentage? ");
    rate = GetReal();
    interest = balance * rate / 100;
    balance = balance + interest;
    printf("Balance after one year: %g\n", balance);
}
```

5.

```
/*
 * File: int2yr.c
 * -----
 * This program calculates compound interest over two years.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
    double balance, rate, interest;

    printf("Interest calculation program.\n");
    printf("Starting balance? ");
    balance = GetReal();
    printf("Annual interest rate percentage? ");
    rate = GetReal();
    interest = balance * rate / 100;
    balance = balance + interest;
    printf("Balance after one year: %g\n", balance);
    interest = balance * rate / 100;
    balance = balance + interest;
    printf("Balance after two years: %g\n", balance);
}
```

6.

```
/*
 * File: circle.c
 * -----
 * This program calculates the area of a circle.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
    double radius, area;

    printf("This program computes the area of a circle.\n");
    printf("What is the radius? ");
    radius = GetReal();
    area = 3.141592 * radius * radius;
    printf("Area = %g\n", area);
}
```

7.

```
/*
 * File: tempconv.c
 * -----
 * This program converts a temperature value given in degrees
 * Fahrenheit into its equivalent in degrees Celsius.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
    double fahrenheit, celsius;

    printf("Program to convert Fahrenheit to Celsius.\n");
    printf("Fahrenheit temperature? ");
    fahrenheit = GetReal();
    celsius = (5.0 / 9.0) * (fahrenheit - 32);
    printf("Celsius equivalent: %g\n", celsius);
}
```

8.

```
/*
 * File: milo.c
 * -----
 * This program computes C's answer to the Mathemagician's
 * question to Milo from the Phantom Tollbooth.
 */

#include <stdio.h>
#include "genlib.h"

main()
{
    int answer;

    answer = 4 + 9 - 2 * 16 + 1 / 3 * 6 - 67 + 8 * 2 - 3 + 26
        - 1 / 34 + 3 / 7 + 2 - 5;
    printf("C's answer to the Mathemagician = %d\n", answer);
}
```

9.

```
/*
 * File: kgtolbs.c
 * -----
 * This program reads in a weight given in kilograms and
 * converts it to its English equivalent in pounds and ounces.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
    double kg, totalPounds, ounces;
    int pounds;

    printf("This program converts kilograms to pounds\n");
    printf("and ounces.\n");
    printf("Weight in kilograms? ");
    kg = GetReal();
    totalPounds = kg * 2.2;
    pounds = (int) totalPounds;
    ounces = (totalPounds - pounds) * 16;
    printf("%g kg = %d lbs %g oz\n", kg, pounds, ounces);
}
```

10.

```
/*
 * File: ave4.c
 * -----
 * This program reads in four integers and computes their
 * average.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
    int n1, n2, n3, n4;
    double average;

    printf("This program averages four integers.\n");
    printf("1st number? ");
    n1 = GetInteger();
    printf("2nd number? ");
    n2 = GetInteger();
    printf("3rd number? ");
    n3 = GetInteger();
    printf("4th number? ");
    n4 = GetInteger();
    average = (double) (n1 + n2 + n3 + n4) / 4;
    printf("The average is %g\n", average);
}
```

11.

```
/*
 * File: stives.c
 * -----
 * This program calculates the number of entities coming
 * from St. Ives in the classic nursery rhyme:
 *
 *   As I was going to St. Ives,
 *   I met a man with seven wives,
 *   Each wife had seven sacks,
 *   Each sack had seven cats,
 *   Each cat had seven kits:
 *   Kits, cats, sacks, and wives,
 *   How many were going to St. Ives?
 */

#include <stdio.h>
#include "genlib.h"

main()
{
    int man, wives, sacks, cats, kits, total;

    man = 1;
    wives = 7 * man;
    sacks = 7 * wives;
    cats = 7 * sacks;
    kits = 7 * cats;
    total = kits + cats + sacks + wives + man;
    printf("Number coming from St. Ives: %d\n", total);
}
```

## Solutions for Chapter 3

### Problem Solving

#### Review questions

1. A *programming idiom* is a pattern that comes up frequently in coding and can be used to accomplish a particular task. Because mastering a set of idioms is generally easier than understanding all the rules that go into making those idioms work, using idioms usually simplifies the learning process.
2. 

```
printf("prompt string");
variable = GetInteger();
```
3. 

```
cellCount *= 2;
```
4. 

```
x++;
```
5. 

```
for (i = 0; i < 15; i++) {
    . . . commands . . .
}
```
6. A *loop* is a section of code that is repeated in a cyclic fashion as part of a program.  
The *control line* is the first line of the loop form and controls how many times the loop is executed.  
The *body* of the loop consists of the statements enclosed in curly braces that are executed repeatedly as specified by the control line.  
A *cycle* is a single execution of the loop body.  
An *index variable* is used in conjunction with a **for** loop and keeps track of the current cycle number.
7. 

```
for (i = 15; i <= 25; i++)
```
8. The assignment to **total** initializes that variable so that it contains 0, which is the correct total before any data values have been read. Without this initialization line, the statement  

```
total += value;
```

inside the loop will add the first value to a variable whose contents are completely unspecified. The variable **value**, however, is assigned a value by the statement  

```
value = GetInteger();
```

before the variable is ever used.
9. A *sentinel* is a special value chosen by the program designer to signify the end of data entry. When you choose a sentinel, you should make sure that it is not a value that might reasonably appear as part of the input data.
10. 

```
while (TRUE) {
    prompt user and read in a value
    if (value == sentinel) break;
    rest of body
}
```
11. The **if** statement appears in this chapter in the following two forms:  

```
if (conditional-test) {
    . . . statements executed if the test is true . . .
}

if (conditional-test) {
    . . . statements executed if the test is true . . .
} else {
    . . . statements executed if the test is false . . .
}
```
12. The six relational operators and corresponding mathematical symbols are

```

==   Equal (=)
!=   Not equal ( )
>    Greater than (>)
<    Less than (<)
>=   Greater than or equal to ( )
<=   Less than or equal to ( )

```

13. Even simple changes that appear to be completely innocuous can easily introduce bugs into a program. Failure to test code thoroughly reduces the likelihood that these bugs will be detected.
14. The format specification `%.2f` is used to display a floating-point number with exactly two digits after the decimal point. This format specification can therefore be used to display a value as dollars and cents.
15. The following statement displays the string `name` in a 20-character, left-justified field:

```
printf("%-20s", name);
```

This statement makes sure that the output never expands beyond 20 characters:

```
printf("%-20.20s", name);
```

16. `printf("%.3f", distance);`
17. You should choose names so that they clearly convey their purpose to the reader. To enhance program readability even more, you should adopt a consistent set of naming conventions so that other programmers reading your code can more easily determine what each name represents.
18. The `#define` construct makes it possible for you to associate a symbolic name with a constant value. Because the symbolic name carries more information, the resulting program is usually easier to read. In addition, defining a constant usually makes it easier to maintain a program by ensuring that a change in the definition of the constant is reflected everywhere that constant is used.
19. `#define Pi 3.14159`
20. The `printf` statement

```
printf("This check bounces.  $%.2f fee deducted.\n",
      BouncedCheckFee);
```

requires `BouncedCheckFee` to be a floating-point value. The `%.2f` format specification means that `printf` will interpret the next argument as a floating-point value. If `BouncedCheckFee` were an integer constant, the result would be unpredictable. It would be easy, however, to solve this problem by using a type cast to ensure that the `printf` call always gets a floating-point value, as follows:

```
printf("This check bounces.  $%.2f fee deducted.\n",
      (double) BouncedCheckFee);
```

21. This review question is open-ended and is intended to encourage creative responses.
22. Your most important audience consists of other programmers who will someday have to read and maintain your code.

## Programming exercises

1.

```
/*
 * File: addn.c
 * -----
 * This program adds a list of n numbers, where n is entered
 * by the user.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
    int n, i, value, total;

    printf("This program adds a list of numbers.\n");
    printf("How many numbers in the list? ");
    n = GetInteger();
    printf("Enter the numbers, one per line.\n");
    total = 0;
    for (i = 0; i < n; i++) {
        printf(" ? ");
        value = GetInteger();
        total += value;
    }
    printf("The total is %d\n", total);
}
```

2.

```
/*
 * File: hello10.c
 * -----
 * This program prints out 10 copies of the "Hello, world."
 * message.
 */

#include <stdio.h>
#include "genlib.h"

main()
{
    int i;

    for (i = 0; i < 10; i++) {
        printf("Hello, world.\n");
    }
}
```



3.

```
/*
 * File: tomorrow.c
 * -----
 * This program prints out the Shakespeare quotation
 * "Tomorrow and tomorrow and tomorrow.".
 */

#include <stdio.h>
#include "genlib.h"

main()
{
    int i;

    printf("Tomorrow");
    for (i = 0; i < 2; i++) {
        printf(" and tomorrow");
    }
    printf(".\n");
}
```

4.

```
/*
 * File: add10f.c
 * -----
 * This program adds a list of ten floating-point numbers,
 * printing the total at the end.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
    double value, total;
    int i;

    printf("This program adds a list of ten numbers.\n");
    total = 0;
    for (i = 0; i < 10; i++) {
        printf(" ? ");
        value = GetReal();
        total += value;
    }
    printf("The total is %g\n", total);
}
```

5.

```
/*
 * File: squares.c
 * -----
 * Program to display a table of squares for the numbers
 * between LowerLimit and UpperLimit, which are defined
 * as constants in the program.
 */

#include <stdio.h>
#include "genlib.h"

/*
 * Constants
 * -----
 * LowerLimit -- Starting value for the table
 * UpperLimit -- Final value for the table
 */

#define LowerLimit 1
#define UpperLimit 10

/* Main program */

main()
{
    int i;

    for (i = LowerLimit; i <= UpperLimit; i++) {
        printf("%d squared is %d\n", i, i * i);
    }
}
```

6.

```
/*
 * File: gauss.c
 * -----
 * This program calculates and displays the sum of the integers
 * between 1 and Max, where Max is a constant.
 */

#include <stdio.h>
#include "genlib.h"

/*
 * Constants
 * -----
 * Max -- Highest integer to include in sum
 */

#define Max 100

/* Main program */

main()
{
    int i, total;

    total = 0;
    for (i = 1; i <= Max; i++) {
        total += i;
    }
    printf("The sum of the integers 1 to %d is %d.\n", Max, total);
}
```

7.

```
/*
 * File: ave5.c
 * -----
 * This program averages a list of integers where the
 * number of values is a constant known in advance.
 * To change the number of values, change the definition
 * of NValues in the "Constants" section.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constants
 * -----
 * NValues -- the number of values to be averaged.
 */

#define NValues 5

/* Main program */

main()
{
    int value, total, i;
    double average;

    printf("This program averages a list of %d integers.\n",
           NValues);
    total = 0;
    for (i = 0; i < NValues; i++) {
        printf(" ? ");
        value = GetInteger();
        total += value;
    }
    average = (double) total / NValues;
    printf("The average is %g\n", average);
}
```

8.

```
/*
 * File: aven.c
 * -----
 * This program averages a list of integers where the
 * number of values is provided by the user.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
    int value, total, i, n;
    double average;

    printf("This program averages a list of integers.\n");
    printf("How many values are there in the list? ");
    n = GetInteger();
    total = 0;
    for (i = 0; i < n; i++) {
        printf(" ? ");
        value = GetInteger();
        total += value;
    }
    average = (double) total / n;
    printf("The average is %g\n", average);
}
```

9.

```
/*
 * File: avelist.c
 * -----
 * This program averages a list of numbers where the
 * end of the input data is indicated by a sentinel.
 * To change the sentinel value, change the definition
 * of Sentinel in the "Constants" section.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constants
 * -----
 * Sentinel -- Value that terminates the input list
 */

#define Sentinel -1

/* Main program */

main()
{
    int value, total, i, n;
    double average;

    printf("This program averages a list of integers.\n");
    printf("Enter %d to signal the end of the list.\n", Sentinel);
    total = 0;
    n = 0;
    while (TRUE) {
        printf(" ? ");
        value = GetInteger();
        if (value == Sentinel) break;
        total += value;
        n += 1;
    }
    average = (double) total / n;
    printf("The average is %g\n", average);
}
```

10.

```
/*
 * File: roses2.c
 * -----
 * This program prints out the Gertrude Stein quotation
 * "a rose is a rose is a rose" but includes the word
 * "rose" only once in the main program.
 */

#include <stdio.h>
#include "genlib.h"

main()
{
    int i;

    for (i = 0; i < 3; i++) {
        printf("a rose");
        if (i < 2) {
            printf(" is ");
        } else {
            printf(".\n");
        }
    }
}
```

11.

```
/*
 * File: cubes.c
 * -----
 * This program displays the squares and cubes of the numbers
 * between LowerLimit and UpperLimit in a tabular form.
 */

#include <stdio.h>
#include "genlib.h"

/*
 * Constants
 * -----
 * LowerLimit -- Starting value for the table
 * UpperLimit -- Final value for the table
 */

#define LowerLimit 1
#define UpperLimit 10

/* Main program */

main()
{
    int i;

    printf("Number Square Cube\n");
    for (i = LowerLimit; i <= UpperLimit; i++) {
        printf(" %2d %3d %4d\n", i, i * i, i * i * i);
    }
}
```

12. The complete program for printing a table of votes is beyond the scope of this chapter. The `printf` line necessary to format the table shown in the text is

```
printf("%-15.15s %4d\n", candidate, votes);
```

13.

```
/*
 * File: intn.c
 * -----
 * This program calculates compound interest over N years.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
    double balance, rate, interest;
    int i, nYears;

    printf("Interest calculation program.\n");
    printf("Starting balance? ");
    balance = GetReal();
    printf("Annual interest rate percentage? ");
    rate = GetReal();
    printf("Number of years? ");
    nYears = GetInteger();
    for (i = 0; i < nYears; i++) {
        interest = balance * rate / 100;
        balance += interest;
        printf("Balance after %d years: %.2f\n", i+1, balance);
    }
}
```



14.

```
/*
 * File: biggest.c
 * -----
 * This program reads in a list of integers and then
 * displays the largest value. The end of the list
 * is indicated by a sentinel. To change the sentinel
 * value, change the definition of Sentinel in the
 * "Constants" section.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constants
 * -----
 * Sentinel -- Value that terminates the input list
 */

#define Sentinel 0

/* Main program */

main()
{
    int largest, current;

    printf("This program finds the largest integer in a list.\n");
    printf("Enter %d to signal the end of the list.\n", Sentinel);
    printf(" ? ");
    largest = GetInteger();
    while (TRUE) {
        printf(" ? ");
        current = GetInteger();
        if (current == Sentinel) break;
        if (current > largest) {
            largest = current;
        }
    }
    printf("The largest value is %d\n", largest);
}
```

## Solutions for Chapter 4

### Statement Forms

#### Review questions

1. The construction is a legal statement but has no useful effect.
2. The statement sets **j** to 4, **k** to 16, and **i** to 20.
3. **x = y = 1.0;**
4. The term *associativity* refers to whether operators of equal precedence are applied from left to right or from right to left. Most operators in C are left-associative; assignment is right-associative.
5. A block is a sequence of statements enclosed in curly braces. Blocks are often called *compound statements* because a block acts syntactically as a statement and can be used in any context in which a statement is required.
6. The two classes of control statements are conditional and iterative.
7. Control statements are said to be nested if one is entirely enclosed by the other.
8. The two values of the data type **bool** are **TRUE** and **FALSE**.
9. In C, the mathematical symbol for equality (=) is used as the assignment operator; testing for equality must be done using the relational operator **==**. Using **=** instead of **==** is a common cause of programming errors because the result is often syntactically legal. Unfortunately, the program performs an assignment at that point instead of testing for equality.
10. The relational operators can only compare atomic objects. They cannot, for example, be used to compare two strings.
11. **n >= 0 && n <= 9**
12. The expression checks to see if the value **x** is either (a) not equal to 4 or (b) not equal to 17. No matter what value **x** has, one of these two conditions must be true because **x** cannot be both 4 and 17 at the same time.
13. *Short-circuit evaluation* means that evaluation of a compound Boolean expression written with the operators **&&** and **||** stops as soon as the result is known.
14. Although the statement given in the text is correct, it is redundant because there is no reason to compare the flag against the constant **TRUE**. Thus, the statement can be simplified to  

```
if (myFlag) . . .
```
15. The four formats used in the text are: (1) the single-line **if** statement, (2) the multiline **if** statement, (3) the **if-else** statement, and (4) the cascading **if** statement.
16. The **switch** statement first evaluates the parenthesized expression, which must be an integer (or integer-like type, as discussed in Chapter 9). The **switch** statement then compares the value against each of the constants appearing in **case** clauses within the body. If a match exists, the statements associated with that **case** clause are executed. If there is no matching value, the statements associated with the **default** clause, if any, are executed instead. If there is no default clause and no value matches the control expression, the **switch** statement does not execute any statements at all.

Each **case** clause ordinarily ends with a **break** statement that causes control to pass to the end of the **switch** statement, and the program continues from there. In the absence of a **break** statement, control continues on to the next **case** clause, even though this condition is likely to represent a programming error.

17. A **while** loop tests the conditional expression only at the beginning of each loop cycle. If the test becomes **FALSE** in the middle of a loop cycle, control continues until that cycle is completed. If the **while** test is still **FALSE** at that point, the loop exits.
18. The **digitsum.c** program uses the remainder operator **%** to select individual digits. The behavior of **%** with negative operands is undefined in C and may produce different results on different machines.
19. The loop-and-a-half problem occurs when the test for loop completion comes logically in the middle of a cycle. One approach to solving this problem is to use the **break** statement to exit from the loop at the point at which the conditional test occurs. It is also possible to reorder the statements in the loop so that the test appears at the beginning, although this approach usually requires some duplication of code.
20. The three expressions in the control line of a **for** statement—*init*, *test*, and *step*—indicate respectively what operations should be performed to initialize the loop, what conditional expression is used to determine whether the loop should continue, and what operations should be performed at the end of one loop cycle to prepare for the next.
21.
  - a. **for** (**i** = 1; **i** <= 100; **i**++)
  - b. **for** (**i** = 0; **i** < 100; **i** += 7)
  - c. **for** (**i** = 100; **i** >= 0; **i** -= 2)
22. Floating-point values are only approximations of real numbers in mathematics and may not always behave the same way. In a **for** loop that tests for equality between two floating-point numbers, slight errors in the computation may cause the loop to execute a different number of cycles than the programmer might expect.

## Programming exercises

1.

```
/*
 * File: beer.c
 * -----
 * This program prints out the lyrics to the song
 * "99 Bottles of Beer on the Wall." For testing
 * purposes, the constant should be reduced to a
 * smaller value.
 */

#include <stdio.h>
#include "genlib.h"

/*
 * Constants
 * -----
 * MaxBottles -- Original number of bottles
 */

#define MaxBottles 99

/* Main program */

main()
{
    int bottles;

    bottles = MaxBottles;
    while (bottles > 0) {
        if (bottles == 1) {
            printf("1 bottle of beer on the wall.\n");
            printf("1 bottle of beer.\n");
        } else {
            printf("%d bottles of beer on the wall.\n", bottles);
            printf("%d bottles of beer.\n", bottles);
        }
        bottles--;
        switch (bottles) {
            case 0:
                printf("No more bottles of beer on the wall.\n");
                break;
            case 1:
                printf("Only one bottle of beer on the wall.\n");
                break;
            default:
                printf("%d bottles of beer on the wall.\n", bottles);
                break;
        }
        printf("\n");
    }
}
```

2.

```
/*
 * File: oldman.c
 * -----
 * This program prints out the lyrics to the song
 * "This old man."
 */

#include <stdio.h>
#include "genlib.h"

main()
{
    int i;
    string rhyme;

    for (i = 1; i <= 10; i++) {
        switch (i) {
            case 1: rhyme = "thumb"; break;
            case 2: rhyme = "shoe"; break;
            case 3: rhyme = "knee"; break;
            case 4: rhyme = "door"; break;
            case 5: rhyme = "hive"; break;
            case 6: rhyme = "sticks"; break;
            case 7: rhyme = "heaven"; break;
            case 8: rhyme = "gate"; break;
            case 9: rhyme = "spine"; break;
            case 10: rhyme = "shin"; break;
        }
        printf("This old man, he played %d.\n", i);
        printf("He played knick-knack on my %s.\n", rhyme);
        printf("With a knick-knack, paddy-whack,\n");
        printf("Give your dog a bone.\n");
        printf("This old man came rolling home.\n");
        printf("\n");
    }
}
```

Unfortunately, the “on my heaven” line in the seventh verse seems a little off. To fix that problem, you could also include a string variable `prep` to hold the appropriate prepositional phrase. For the seventh verse, `prep` would be set to “up to”; for all the others `prep` would be set to “on my”. The second line of the song could then be displayed using the statement

```
printf("He played knick-knack %s %s.\n", prep, rhyme);
```

3.

```
/*
 * File: oddsum.c
 * -----
 * This program calculates the sum of the first N odd integers.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Implementation notes
 * -----
 * This implementation uses a variable odd to keep track
 * of each odd number and adds two to it on each cycle
 * to get the next one. Alternatively, you could use
 * the fact that the ith odd number is given by
 *
 *   2 * i - 1
 *
 * and use that formula in place of the variable odd.
 */

main()
{
    int i, n, total, odd;

    printf("This program sums the first N odd integers.\n");
    printf("N = ? ");
    n = GetInteger();
    total = 0;
    odd = 1;
    for (i = 0; i < n; i++) {
        total += odd;
        odd += 2;
    }
    printf("The sum of the first %d odd integers is %d\n", n, total);
}
```

This program, however, can be written much more efficiently by taking advantage of a little mathematics. The following implementation calculates the same result with far fewer operations.

```
/*
 * File: oddsum2.c
 * -----
 * This program computes the sum of the first N odd integers by
 * exploiting the simple mathematical relationship that this sum
 * is equal to N squared. Using mathematics is not "cheating."
 * The resulting program will behave in exactly the same way as
 * far as the user is concerned but computes the results much more
 * efficiently.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Implementation notes
 * -----
 * This implementation is considerably more efficient than
 * the one using a for loop to sum the odd integers. In this
 * case, the implementation takes advantage of the mathematical
 * property that the sum of the first n odd integers is equal
 * to n squared. This property can be proved formally using a
 * technique called mathematical induction, but it can also be
 * verified less formally using the following geometric argument:
 *
 *      1 3 5 7
 *      3 3 5 7
 *      5 5 5 7
 *      7 7 7 7
 *
 * The figure has one 1, three 3s, five 5s, and each new odd
 * integer can be arranged to form the next larger square.
 */

main()
{
    int n;

    printf("This program sums the first N odd integers.\n");
    printf("N = ? ");
    n = GetInteger();
    printf("The sum of the first %d odd integers is %d\n", n, n * n);
}
```

4.

```
/*
 * File: div6or7.c
 * -----
 * This program lists all numbers between LowerLimit and
 * UpperLimit that are divisible by 6 or 7.
 */

#include <stdio.h>
#include "genlib.h"

/*
 * Constants
 * -----
 * LowerLimit -- Lowest value to check
 * UpperLimit -- Highest value to check
 */

#define LowerLimit 1
#define UpperLimit 100

/* Main program */

main()
{
    int i;

    printf("This program lists the numbers between %d and %d\n",
           LowerLimit, UpperLimit);
    printf("that are divisible by 6 or 7.");
    for (i = LowerLimit; i <= UpperLimit; i++) {
        if ((i % 6 == 0) || (i % 7 == 0)) {
            printf("%3d\n", i);
        }
    }
}
```



5.

```
/*
 * File: div6xor7.c
 * -----
 * This program lists all numbers between LowerLimit and
 * UpperLimit that are divisible by 6 or 7, but not both.
 */

#include <stdio.h>
#include "genlib.h"

/*
 * Constants
 * -----
 * LowerLimit -- Lowest value to check
 * UpperLimit -- Highest value to check
 */

#define LowerLimit 1
#define UpperLimit 100

/* Main program */

/*
 * Implementation notes
 * -----
 * There are several ways to solve this problem. The most
 * straightforward is to write the complete Boolean
 * expression that corresponds to the compound condition
 *
 * ((divisible by 6) or (divisible by 7))
 * and not ((divisible by 6) and (divisible by 7))
 *
 * which is the condition used in the program shown.
 * You can, however, use an even simpler form if you
 * recognize that the desired condition occurs only
 * when the result of the conditions (i % 6 == 0) and
 * (i % 7 == 0) are different: one is TRUE and the other
 * FALSE. Given this insight, you could use the following
 * if test instead:
 *
 * if ((i % 6 == 0) != (i % 7 == 0))
 */

main()
{
    int i;

    printf("This program lists the numbers between %d and %d\n",
           LowerLimit, UpperLimit);
    printf("that are divisible by 6 or 7, but not both.");
    for (i = LowerLimit; i <= UpperLimit; i++) {
        if (((i % 6 == 0) || (i % 7 == 0))
            && !((i % 6 == 0) && (i % 7 == 0))) {
            printf("%3d\n", i);
        }
    }
}
```

6.

```
/*
 * File: liftoffw.c
 * -----
 * This program simulates a countdown for a rocket launch, just
 * like the liftoff program from the text. The only difference
 * is that this implementation uses a while loop instead of a
 * for loop.
 */

#include <stdio.h>
#include "genlib.h"

/*
 * Constant: StartingCount
 * -----
 * Change this constant to use a different starting value
 * for the countdown.
 */

#define StartingCount 10

/* Main program */

main()
{
    int t;

    t = StartingCount;
    while (t >= 0) {
        printf("%2d\n", t);
        t--;
    }
    printf("Liftoff!\n");
}
```

7.

```
/*
 * File: revdigit.c
 * -----
 * This program generates and displays the integer formed
 * by reversing the digits in another integer. For example,
 * given the input 1729, this program would display the
 * integer 9271.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
    int n, rev;

    printf("This program reverses the digits in an integer.\n");
    printf("Enter a positive integer: ");
    n = GetInteger();
    rev = 0;
    while (n > 0) {
        rev = (10 * rev) + n % 10;
        n /= 10;
    }
    printf("The reversed number is %d\n", rev);
}
```

8.

```
/*
 * File: fibn.c
 * -----
 * This program lists the terms in the Fibonacci sequence from
 * F(0) to F(MaxIndex).
 */

#include <stdio.h>
#include "genlib.h"

/*
 * Constants
 * -----
 * MaxIndex -- Highest term index to generate
 */

#define MaxIndex 15

/* Main program */

/*
 * Implementation notes
 * -----
 * In this implementation, the program keeps track of
 * three consecutive terms in the variables t1, t2, and t3.
 * On each loop cycle, the next term (t3) is computed by
 * adding t1 and t2. As soon as it does so, it prepares
 * for the next cycle, by shifting the terms so that the
 * previous t2 becomes the new t1 and the previous t3
 * becomes t2.
 */

main()
{
    int t1, t2, t3, i;

    printf("This program lists the Fibonacci sequence.\n");
    t1 = 0;
    t2 = 1;
    for (i = 0; i <= MaxIndex; i++) {
        if (i < 10) printf(" ");
        printf("F(%d) = %4d\n", i, t1);
        t3 = t1 + t2;
        t1 = t2;
        t2 = t3;
    }
}
```

9.

```
/*
 * File: fibmax.c
 * -----
 * This program lists the terms in the Fibonacci sequence until
 * the value of the current term exceeds MaxFib.
 */

#include <stdio.h>
#include "genlib.h"

/*
 * Constants
 * -----
 * MaxFib -- Largest value to generate
 */

#define MaxFib 10000

/* Main program */

/*
 * Implementation notes
 * -----
 * In this implementation, the program keeps track of
 * three consecutive terms in the variables t1, t2, and t3.
 * On each loop cycle, the next term (t3) is computed by
 * adding t1 and t2. As soon as it does so, it prepares
 * for the next cycle, by shifting the terms so that the
 * previous t2 becomes the new t1 and the previous t3
 * becomes t2.
 */

main()
{
    int t1, t2, t3, i;

    printf("This program lists the Fibonacci sequence.\n");
    t1 = 0;
    t2 = 1;
    for (i = 0; t1 <= MaxFib; i++) {
        if (i < 10) printf(" ");
        printf("F(%d) = %4d\n", i, t1);
        t3 = t1 + t2;
        t1 = t2;
        t2 = t3;
    }
}
```

10.

```
/*
 * File: triangle.c
 * -----
 * This program displays a triangular pattern of stars.
 */

#include <stdio.h>
#include "genlib.h"

/*
 * Constants
 * -----
 * NRows -- Number of rows in the figure.
 */

#define NRows 8

/* Main program */

main()
{
    int i, j;

    for (i = 1; i <= NRows; i++) {
        for (j = 0; j < i; j++) {
            printf("*");
        }
        printf("\n");
    }
}
```

11.

```
/*
 * File: uptriang.c
 * -----
 * This program displays a triangular pattern with the
 * point facing upward.
 */

#include <stdio.h>
#include "genlib.h"

/*
 * Constants
 * -----
 * NRows -- Number of rows in the figure.
 */

#define NRows 8

/* Main program */

main()
{
    int i, j;

    for (i = 1; i <= NRows; i++) {
        for (j = i; j < NRows; j++) {
            printf(" ");
        }
        for (j = 0; j < 2 * i - 1; j++) {
            printf("*");
        }
        printf("\n");
    }
}
```

## Solutions for Chapter 5

### Functions

#### Review questions

1. A *function* is a named collection of statements that performs a particular operation. A *program* is a collection of functions, including one called **main**, that together execute a task on behalf of the user.
2. Whenever you need to invoke the operation performed by a particular function, you *call* that function by specifying its name, followed by a list of values enclosed in parentheses. When the computer executes the function call, the original function is suspended at that point, and the computer continues execution beginning with the first statement in the called function. The values in parentheses are called *arguments*; these values are copied into the corresponding formal parameter variables declared by the function. When the operation of the function is complete, that function *returns* to the calling program, which means that the completed function and all of its variables disappear, after which the computer continues executing the previously suspended calling function.
3. Arguments provide a mechanism by which one function can communicate information to another. Input functions like **GetInteger**, on the other hand, provide a mechanism by which the user can communicate information to a program. If, for example, a program were designed to add a list of integer entered by the user, those values would be read using **GetInteger**. If, however, you needed to calculate a square root as part of some more complex computation, you would instead pass the appropriate value as an argument to the **sqrt** function.
4. **double sqrt(double);**
5. From the prototype alone, you know that the **atan2** function takes two values of type **double** as arguments and returns a value that is also of type **double**.
6. Parameter names in a function prototype convey extra information to the reader of the program about what those parameters represent.
7. A function that takes no arguments uses the keyword **void**, as in  

```
int GetInteger(void);
```
8. To return a value from a function, you use the keyword **return** followed by an expression specifying the return value.
9. There can be any number of **return** statements in a function.
10. The **case** clauses in the **MonthName** function do not require **break** statements because they end with a **return** statement. The **return** statement causes the program to exit at that point, which means that the program will not continue into the next **case** clause.
11. A predicate function is one that returns a Boolean value.
12. The **StringEqual** function defined in **string.h** allows you to test whether two strings are equal.
13. In this text, the term *arguments* is used to refer to the expressions that are part of a function call; the term *formal parameters* refers to the variable names in the function header into which the argument values are copied.
14. Local variables are only visible inside the function in which they are declared. Outside that function, those variables are undefined, which means that their declaration is “local” to the defining function.
15. The term *return address* signifies the point in the calling function at which execution will continue when a function returns. The computer must save the return address whenever it makes a function call.
16. A procedure is a function that returns no result.



17. The strategy of stepwise refinement consists of breaking a program down into functions that solve successively simpler subproblems. To apply the technique, you always begin with the most general statement of the problem. You code the solution to the problem as a function in which you represent any complex operations as function calls, even though you have yet to write the definitions of those functions. Once you finish one level of the decomposition, you can go ahead and apply the same process to the functions that solve the subproblems, which may in turn divide to form new subproblems. The process continues until all operations are simple enough to code without further subdivision.

### Programming exercises

- 1.
- ```
/*
 * File: golden.c
 * -----
 * This program calculates the golden ratio.
 */

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

main()
{
    double phi;

    phi = (1 + sqrt(5)) / 2;
    printf("The golden ratio is %g\n", phi);
}
```

2.

```

/*
 * File: quadeq.c
 * -----
 * This program finds roots of the quadratic equation
 *
 *      2
 *    a x  + b x + c = 0
 *
 * If there are no real roots, the program prints a message
 * to that effect.
 */

#include <stdio.h>
#include <math.h>
#include "genlib.h"
#include "simpio.h"

main()
{
    double a, b, c, disc, sqrtDisc, x1, x2;

    printf("Enter coefficients for the quadratic equation:\n");
    printf("a: ");
    a = GetReal();
    printf("b: ");
    b = GetReal();
    printf("c: ");
    c = GetReal();
    disc = b * b - 4 * a * c;
    if (disc < 0) {
        printf("The solutions are complex numbers.\n");
    } else {
        sqrtDisc = sqrt(disc);
        x1 = (-b + sqrtDisc) / (2 * a);
        x2 = (-b - sqrtDisc) / (2 * a);
        printf("The first solution is %g\n", x1);
        printf("The second solution is %g\n", x2);
    }
}

```

A common mistake in writing this function is to forget the parentheses in the denominator and write lines like

```
x1 = (-b + sqrtDisc) / 2 * a;
```

instead of

```
x1 = (-b + sqrtDisc) / (2 * a);
```

This problem, however, often goes unnoticed when the program is tested. Many students only use test data with **a** equal to 1 because the answers are usually easier to compute; when **a** is 1, the problem does not arise. Other students, however, will use values for **a** other than 1 and still fail to notice the error. Even though the program delivers an incorrect answer, many students never bother to check. This problem can therefore be used as a good object lesson for the importance of taking testing seriously.

3.

```
/*
 * File: f2ctable.c
 * -----
 * This program is a simple adaptation of the Celsius
 * to Fahrenheit conversion program in the text. It
 * converts in the opposite direction.
 */

#include <stdio.h>
#include "genlib.h"

/*
 * Constants
 * -----
 * LowerLimit -- Starting value for temperature table
 * UpperLimit -- Final value for temperature table
 * StepSize   -- Step size between table entries
 */

#define LowerLimit 32
#define UpperLimit 100
#define StepSize 2

/* Function prototypes */

double FahrenheitToCelsius(double c);

/* Main program */

main()
{
    int f;

    printf("Fahrenheit to Celsius table.\n");
    printf(" F      C\n");
    for (f = LowerLimit; f <= UpperLimit; f += StepSize) {
        printf("%3d  %5.1f\n", f, FahrenheitToCelsius(f));
    }
}

/*
 * Function: FahrenheitToCelsius
 * Usage: c = FahrenheitToCelsius(f);
 * -----
 * This function returns the Celsius equivalent of the Fahrenheit
 * temperature f.
 */

double FahrenheitToCelsius(double f)
{
    return (5.0 / 9.0 * (f - 32));
}
```

4.

```
/*
 * File: fibfn.c
 * -----
 * This program lists the terms in the Fibonacci sequence from
 * Fib(0) to Fib(MaxIndex). This implementation uses a function
 * to calculate each term. Because the calculation of each new
 * term makes no use of the preceding ones, this implementation
 * is considerably less efficient than the one given in Chapter 4.
 */

#include <stdio.h>
#include "genlib.h"

/*
 * Constants
 * -----
 * MaxIndex -- Highest term index to generate
 */

#define MaxIndex 15

/* Function prototypes */

int Fib(int n);

/* Main program */

main()
{
    int i;

    printf("This program lists the Fibonacci sequence.\n");
    for (i = 0; i <= MaxIndex; i++) {
        if (i < 10) printf(" ");
        printf("F(%d) = %4d\n", i, Fib(i));
    }
}

/*
 * Function: Fib
 * Usage: t = Fib(n);
 * -----
 * This function returns the nth term in the Fibonacci sequence.
 */

int Fib(int n)
{
    int t1, t2, t3, i;

    t1 = 0;
    t2 = 1;
    for (i = 0; i < n; i++) {
        t3 = t1 + t2;
        t1 = t2;
        t2 = t3;
    }
    return (t1);
}
```

5.

```
/*
 * File: intexp.c
 * -----
 * This program includes the RaiseIntToPower function, along
 * with a test program that displays powers of two.
 */

#include <stdio.h>
#include "genlib.h"

/*
 * Constants
 * -----
 * LowerLimit -- Starting value for the table
 * UpperLimit -- Final value for the table
 */

#define LowerLimit 0
#define UpperLimit 10

/* Function prototypes */

int RaiseIntToPower(int n, int k);

/* Main program */

main()
{
    int k;

    printf("          k\n");
    printf(" k          2\n");
    printf("-----\n");
    for (k = LowerLimit; k <= UpperLimit; k++) {
        printf("%2d    %4d\n", k, RaiseIntToPower(2, k));
    }
}

/*
 * Function: RaiseIntToPower
 * Usage: p = RaiseIntToPower(n, k);
 * -----
 * This function returns the integer n raised to the kth power.
 */

int RaiseIntToPower(int n, int k)
{
    int i, result;

    result = 1;
    for (i = 0; i < k; i++) {
        result *= n;
    }
    return (result);
}
```

6.

```

/*
 * File: realexp.c
 * -----
 * This program includes the RaiseRealToPower function, along with a
 * test program that prints powers of ten (including negative powers).
 */

#include <stdio.h>
#include <stdlib.h>
#include "genlib.h"

/*
 * Constants
 * -----
 * LowerLimit -- Starting value for the table
 * UpperLimit -- Final value for the table
 */

#define LowerLimit -4
#define UpperLimit 4

/* Function prototypes */

double RaiseRealToPower(double x, int k);

/* Main program */

main()
{
    int k;

    printf("          k\n");
    printf(" k          10\n");
    printf("-----\n");
    for (k = LowerLimit; k <= UpperLimit; k++) {
        if (k < 0) {
            printf("%2d          %g\n", k, RaiseRealToPower(10, k));
        } else {
            printf("%2d      %7.1f\n", k, RaiseRealToPower(10, k));
        }
    }
}

/*
 * Function: RaiseRealToPower
 * Usage: p = RaiseRealToPower(x, k);
 * -----
 * This function returns the real number x raised to the
 * kth power. The integer k may be negative.
 */

double RaiseRealToPower(double x, int k)
{
    int i;
    double result;

    result = 1;

    for (i = 0; i < abs(k); i++) {
        result *= x;
    }
    if (k < 0) result = 1.0 / result;
    return (result);
}

```

7.

```
/*
 * File: ndigits.c
 * -----
 * This program defines the function NDigits, along with
 * a simple main program to test the function.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/* Function prototypes */

int NDigits(int n);

/* Main program */

main()
{
    int n;

    printf("This program tests the NDigits function.\n");
    printf("Enter integers, using any negative value to stop.\n");
    while (TRUE) {
        printf(" ? ");
        n = GetInteger();
        if (n < 0) break;
        printf("Number of digits = %d\n", NDigits(n));
    }
}

/*
 * Function: NDigits
 * Usage: nd = NDigits(n);
 * -----
 * This function returns the number of digits in the integer n.
 * The value is obtained by counting how many times n can be
 * divided by 10 before becoming zero.
 */

int NDigits(int n)
{
    int nd;

    nd = 1;
    while (n / 10 != 0) {
        n /= 10;
        nd++;
    }
    return (nd);
}
```

8.

```
/*
 * Function: Round
 * Usage: n = Round(x);
 * -----
 * This function returns the closest integer to x.
 */

int Round(double x)
{
    if (x < 0) {
        return ((int) (x - 0.5));
    } else {
        return ((int) (x + 0.5));
    }
}
```



9.

```
/*
 * File: issquare.c
 * -----
 * This program defines the function IsPerfectSquare, along
 * with a test program that lists the perfect squares from
 * 1 to 1000.
 */

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

/*
 * Constants
 * -----
 * LowerLimit -- Starting value for the table
 * UpperLimit -- Final value for the table
 */

#define LowerLimit    1
#define UpperLimit 1000

/* Function prototypes */

bool IsPerfectSquare(int n);
int Round(double x);

/* Main program */

main()
{
    int n;

    printf("This program lists the perfect squares between");
    printf(" %d and %d.\n", LowerLimit, UpperLimit);
    for (n = LowerLimit; n <= UpperLimit; n++) {
        if (IsPerfectSquare(n)) {
            printf("%4d\n", n);
        }
    }
}

/*
 * Function: IsPerfectSquare
 * Usage: if (IsPerfectSquare(n)) . . .
 * -----
 * This function returns TRUE if n is a perfect square.
 */

bool IsPerfectSquare(int n)
{
    int root;

    root = Round(sqrt(n));
    return (n == root * root);
}
```

10.

```

/*
 * Function: ApproximatelyEqual
 * Usage: if (ApproximatelyEqual(x, y)) . . .
 * -----
 * This function returns TRUE if x and y are approximately
 * equal, as indicated by the formula:
 *
 *      | x - y |
 *      ----- < Epsilon
 *      min(|x|, |y|)
 *
 * To avoid the possibility of division by 0, the function
 * first tests to make sure that adding the denominator to
 * the numerator of this fraction changes the nominator.
 * This test has the same practical effect as checking
 * against 0 but avoids overflow in the case that the
 * denominator is very small.
 */

bool ApproximatelyEqual(double x, double y)
{
    double num, den;

    num = fabs(x - y);
    den = MinF(fabs(x), fabs(y));
    if (num + den == num) return (x == y);
    return (num / den < Epsilon);
}

/*
 * Function: MinF
 * Usage: min = MinF(x, y);
 * -----
 * This function returns the smaller of the two floating-point
 * values x and y.
 */

double MinF(double x, double y)
{
    if (x < y) {
        return (x);
    } else {
        return (y);
    }
}

```

In this problem, it is important for students to recognize that the program must make a special check to see whether the denominator of the formula is zero, even though this fact is not explicitly specified in the problem. Because most programming problems are incompletely specified, programmers are forced to be alert to potential hazards of this sort.

11.

```
/*
 * Function: GetYesOrNo
 * Usage: if (GetYesOrNo(prompt)) . . .
 * -----
 * This function asks the user the question indicated by prompt
 * and waits for a yes/no response.  If the user answers "yes"
 * or "no", the program returns TRUE or FALSE accordingly.
 * If the user gives any other response, the program asks
 * the question again.
 */

bool GetYesOrNo(string prompt)
{
    string answer;

    while (TRUE) {
        printf("%s", prompt);
        answer = GetLine();
        if (StringEqual(answer, "yes")) return (TRUE);
        if (StringEqual(answer, "no")) return (FALSE);
        printf("Please answer yes or no.\n");
    }
}
```

12.

```
/*
 * File: pascal.c
 * -----
 * This program generates the Pascal Triangle for rows
 * numbered 0 to MaxN.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constants
 * -----
 * MaxN -- Maximum value for n in C(n,k)
 */

#define MaxN 7

/* Function prototypes */

void PrintTriangleRow(int n);
void PrintLeadingSpaceForRow(int n);
int Combinations(int n, int k);
int Factorial(int n);

/* Main program */

main()
{
    int n;

    for (n = 0; n <= MaxN; n++) {
        PrintTriangleRow(n);
    }
}

/*
 * Function: PrintTriangleRow
 * Usage: PrintTriangleRow(n)
 * -----
 * This function prints the nth row of Pascal's triangle.
 */

void PrintTriangleRow(int n)
{
    int k;

    PrintLeadingSpaceForRow(n);
    for (k = 0; k <= n; k++) {
        printf(" %2d ", Combinations(n, k));
    }
    printf("\n");
}
```

```
/*
 * Function: PrintLeadingSpaceForRow
 * Usage: PrintLeadingSpaceForRow(n)
 * -----
 * Pascal's triangle widens as it goes down, so that the
 * early rows in the triangle need to start with extra
 * spaces before the actual row begins. Row MaxN starts
 * at the margin, and therefore needs no extra space.
 * For each row we go back in the table, we need to indent
 * the line half the width of a complete field. Since the
 * printf format string " %2d " in PrintTriangleRow indicates
 * a field width of four, we need to print two spaces for
 * each row we happen to be from the last row.
 */

void PrintLeadingSpaceForRow(int n)
{
    int i;

    for (i = n; i < MaxN; i++) {
        printf("  ");
    }
}

/*
 * Function: Combinations
 * Usage: ways = Combinations(n, k);
 * -----
 * This function implements the Combinations function, which
 * returns the number of distinct ways of choosing k objects
 * from a set of n objects. In mathematics, this function is
 * often written as C(n,k).
 */

int Combinations(int n, int k)
{
    return (Factorial(n) / (Factorial(k) * Factorial(n - k)));
}

/*
 * Function: Factorial
 * Usage: f = Factorial(n);
 * -----
 * This function returns the factorial of the argument n, which
 * is defined as the product of all integers from 1 to n.
 */

int Factorial(int n)
{
    int product, i;

    product = 1;
    for (i = 1; i <= n; i++) {
        product *= i;
    }
    return (product);
}
```

13. In the on-line solutions library, the solutions for exercises 13 and 14 are combined into a single program `newcal.c` that implements both extensions. For exercise 13, the only part of the program that needs to change is the `GetYearFromUser` function, which looks like this:

```
/*
 * Function: GetYearFromUser
 * Usage: year = GetYearFromUser();
 * -----
 * This function reads in a year from the user and returns
 * that value.  If the user enters a year in the range 0-99,
 * the year is assumed to be in the 20th century.
 */

int GetYearFromUser(void)
{
    int year;

    printf("Which year? ");
    year = GetInteger();
    if (0 <= year && year < 100) year += 1900;
    return (year);
}
```

If exercise 13 is implemented without the further generalization from exercise 14, this code should retain the checks for out-of-range years included in the textbook version of the code.

13. To accommodate years prior to 1900, the following changes must be made to the `FirstDayOfMonth` function:

```
/*
 * Function: FirstDayOfMonth
 * Usage: weekday = FirstDayOfMonth(month, year);
 * -----
 * This function returns the day of the week on which the
 * indicated month begins. This program simply counts either
 * forward or backward from January 1, 1900, which was a
 * Monday. Because  $365 \% 7$  is 1, each preceding year begins
 * one weekday earlier (two if it's a leap year). Adding six
 * gives the same weekday as subtracting one but is always
 * positive and therefore avoids using  $\%$  with negative values.
 */

int FirstDayOfMonth(int month, int year)
{
    int weekday, i;

    weekday = Monday;
    if (year >= 1900) {
        for (i = 1900; i < year; i++) {
            weekday = (weekday + 365) % 7;
            if (IsLeapYear(i)) weekday = (weekday + 1) % 7;
        }
    } else {
        for (i = year; i < 1900; i++) {
            weekday = (weekday + 6) % 7;
            if (IsLeapYear(i)) weekday = (weekday + 6) % 7;
        }
    }
    for (i = 1; i < month; i++) {
        weekday = (weekday + MonthDays(i, year)) % 7;
    }
    return (weekday);
}
```

## Solutions for Chapter 6

### Algorithms

#### Review questions

1. For a solution technique to be an algorithm, it must be
  - a. Clearly and unambiguously defined
  - b. Effective, in the sense that its steps are executable
  - c. Finite, in the sense that it terminates after a bounded number of steps
2. A loop invariant is any property that is true before executing every cycle of a loop.
3. If a number  $n$  has a factor  $f$ , there must, by definition, be some integer  $g$  such that

$$f \times g = n$$

At least one of  $f$  or  $g$  must be less than or equal to the square root of  $n$ . If both were larger, then the product would end up being too large. Thus, if  $n$  has a factor greater than the square root of  $n$ , it must also have one that is smaller than the square root of  $n$ . If your only goal is to find out whether  $n$  has any factors at all, you need only check possibilities up to the square root.

4. When you use a loop within a program, check to see if there are any calculations within the loop that could just as well be performed before the loop begins. If there are, you can increase the efficiency of the program by calculating the result once, storing the result in a variable, and then using that variable inside the loop.
5. The primary concern in choosing an algorithm is correctness. After that, you should consider the tradeoffs between other factors including efficiency, clarity, and maintainability.
6. Euclid's algorithm works even if  $x$  is smaller than  $y$ . When this occurs, the first cycle of the **while** loop ends up exchanging the values of  $x$  and  $y$ .
7. There are several techniques you can use to test whether a solution has converged. One approach is to define some small constant and wait until the solution comes within that distance of the actual solution. In many cases, it is possible to continue the process until the result stops getting closer, which indicates that you have reached the limit of machine accuracy.
8. The main difference between **Error** and **printf** is that **printf** returns to its caller while **Error** simply stops after printing the message. The other differences are that **Error** prints the string "**Error:** " before the message and automatically includes a newline character at the end of the line.
9. In the text, Zeno's paradox is presented in the context of trying to walk across a room. To cross the room, one must first get to the point halfway across. From there, one must again go half the remaining distance, and so on. Zeno argued that, since there is always some distance remaining that can be halved, motion is impossible. Zeno's paradox is more often illustrated using the parable of Achilles and the Tortoise. Suppose that Achilles—the fastest runner alive—is trying to catch the Tortoise, who has a head start. Achilles runs to where the Tortoise started, but the Tortoise has by then moved a little farther along. Achilles then races to the new point, by which time the Tortoise has again managed to make additional progress. In this case, Zeno's argument is that Achilles can never catch up to the Tortoise.
10. In most cases, it is better to compute each term from the previous one because there are fewer computations involved.
11. Even though the two control lines are mathematically identical, floating-point numbers are only approximations of real numbers in mathematics. The test

```
sum != sum + term
```

checks to see if the value of **term** is so small in comparison to **sum** that adding it in does not change the result. For example, if the value of **sum** is 2.3 and the value of **term** is .000000000000000001, on most machines it is



likely that `sum + term` will end up being simply 2.3. On the other hand, `term` is not equal to 0. Thus, a program that used the first `while` test would terminate at this point, whereas a program that used the second test would continue to run.

12. The value 0 is outside the radius of convergence for the Taylor series, which works only for values of  $x$  that fall in the range  $0 < x < 2$ .

### Programming exercises

1.

```
/*
 * Function: SolveMaze
 * Usage: SolveMaze();
 * -----
 * This function implements the right-hand rule algorithm for
 * solving a maze, using as primitives the following functions:
 *
 * MoveForward()  Move forward into the next square
 * TurnRight()   Turn right without moving
 * TurnLeft()    Turn left without moving
 * IfFacingWall() TRUE if Theseus is facing a wall
 * IfOutside()   TRUE if Theseus has escaped
 */

void SolveMaze(void)
{
    while (!IfOutside()) {
        TurnRight();
        while (IfFacingWall()) {
            TurnLeft();
        }
        MoveForward();
    }
}
```

The on-line solutions file also contains a library package called `mazemap` that implements the required maze solving operations using the extended graphics library available from the `aw.com` FTP site. This package allows students to watch their programs run on the display screen.

2.

```
/*
 * File: factor.c
 * -----
 * This program decomposes a number into its prime factors.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/* Function prototypes */

void PrintFactors(int n);

/* Main program */

main()
{
    int n;

    printf("Enter number to be factored: ");
    n = GetInteger();
    PrintFactors(n);
}

/*
 * Function: PrintFactors
 * Usage: PrintFactors(n);
 * -----
 * This function displays the factorization of a number n.
 */

void PrintFactors(int n)
{
    int factor;
    bool first;

    if (n < 2) Error("Number must be greater than 1");
    first = TRUE;
    for (factor = 2; n > 1; factor++) {
        while (n % factor == 0) {
            if (! first) printf(" * ");
            printf("%d", factor);
            first = FALSE;
            n /= factor;
        }
    }
    printf("\n");
}
```

3.

```
/*
 * File: perfect.c
 * -----
 * This file implements a brute-force solution to the perfect
 * number problem.
 */

#include <stdio.h>
#include "genlib.h"

/*
 * Constants
 * -----
 * LowerLimit -- Starting value for the prime search
 * UpperLimit -- Final value for the prime search
 */

#define LowerLimit 1
#define UpperLimit 9999

/* Function prototypes */

bool IsPerfect(int n);

/* Main program */

main()
{
    int i;

    for (i = LowerLimit; i <= UpperLimit; i++) {
        if (IsPerfect(i)) {
            printf("%4d\n", i);
        }
    }
}

/*
 * Function: IsPerfect
 * Usage: if (IsPerfect(n)) . . .
 * -----
 * This function returns TRUE if n is a perfect number. The
 * implementation simply checks every possible number to see
 * if it is a divisor, adding all of the successful candidates
 * together.
 */

bool IsPerfect(int n)
{
    int i, sum;

    sum = 0;
    for (i = 1; i < n; i++) {
        if (n % i == 0) sum += i;
    }
    return (sum == n);
}
```

4. There are many improvements that can be made to the **IsPerfect** implementation given in exercise 3. The following implementation reduces the required computation time substantially without requiring any sophisticated mathematics:

```
/*
 * Function: IsPerfect
 * Usage: if (IsPerfect(n)) . . .
 * -----
 * This function returns TRUE if n is a perfect number.
 * This implementation takes advantage of the observation
 * that all factors less than the square root of n are
 * paired with a factor larger than the square root of
 * of n, and that IsPerfect can add in both factors at
 * once. Note that there does need to be a check in the
 * case of a perfect square to ensure that the square root
 * is not counted twice. It is also much more important
 * for this function to ensure that the loop does not go
 * past the square root, which might result in double-
 * counting. Thus, this function finds the integer square
 * root by adding in a small adjustment value Epsilon
 * before truncating to an integer. The function also
 * returns immediately if the sum ever exceeds n.
 */

bool IsPerfect(int n)
{
    int i, sum, limit;

    if (n < 2) return (FALSE);
    sum = 1;
    limit = sqrt(n) + Epsilon;
    for (i = 2; i <= limit; i++) {
        if (n % i == 0) {
            sum += i + n / i;
            if (sum > n) return (FALSE);
        }
    }
    if (limit * limit == n) sum += limit;
    return (sum == n);
}
```

5.

```
/*
 * Function: CubeRoot
 * Usage: root = CubeRoot(x);
 * -----
 * This function returns the cube root of x, calculated using
 * a modified version of Newton's algorithm as presented in the
 * text. The program makes use of the ApproximatelyEqual
 * function from Chapter 5, exercise 10.
 */

double CubeRoot(double x)
{
    double g;

    if (x == 0) return (0);
    g = x;
    while (!ApproximatelyEqual(x, g * g * g)) {
        g = (g + x / (g * g)) / 2;
    }
    return (g);
}
```

6.

```

/*
 * File: planets.c
 * -----
 * This program uses Bode's law to compute planetary distances.
 */

#include <stdio.h>
#include "genlib.h"

/* Constants */

#define NPlanets 8 /* Number of planets (including asteroids) */

/* Function prototypes */

string PlanetName(int k);

/* Main program */

main()
{
    double base, distance;
    int i;

    for (i = 1; i <= NPlanets; i++) {
        switch (i) {
            case 1: base = 1; break;
            case 2: base = 3; break;
            default: base *= 2; break;
        }
        distance = (base + 4) / 10;
        printf("%-10s %5.1f AU\n", PlanetName(i), distance);
    }
}

/*
 * Function: PlanetName
 * Usage: name = PlanetName(k);
 * -----
 * This function returns the name of the kth planet.
 */

string PlanetName(int k)
{
    switch (k) {
        case 1: return ("Mercury");
        case 2: return ("Venus");
        case 3: return ("Earth");
        case 4: return ("Mars");
        case 5: return ("Asteroids");
        case 6: return ("Jupiter");
        case 7: return ("Saturn");
        case 8: return ("Uranus");
        default: Error("No such planet was known in 1772");
    }
}

```

7.

```
/*
 * File: pil.c
 * -----
 * This program calculates an approximation to pi
 * by summing the series
 *
 *      1      1      1      1      1
 *      - - - + - - - - - + - - - - - + ...
 *      3      5      7      9     11
 *
 */

#include <stdio.h>
#include "genlib.h"

/*
 * Constants
 * -----
 * NTerms -- Number of terms to compute
 */

#define NTerms 10000

/* Main program */

main()
{
    double sum, denom;
    int sign, i;

    sum = 0;
    denom = 1;
    sign = 1;
    for (i = 0; i < NTerms; i++) {
        sum += sign / denom;
        denom += 2;
        sign = -sign;
    }
    printf("Pi is close to %g\n", sum * 4);
}
```

8.

```
/*
 * File: pi2.c
 * -----
 * This program calculates a better approximation to pi
 * by summing the series given in the text.
 */

#include <stdio.h>
#include "genlib.h"

/* Main program */

main()
{
    double sum, term, coeff, hpower;
    int i;

    coeff = 1;
    sum = 0;
    term = hpower = 0.5;
    for (i = 1; sum != sum + term; i += 2) {
        sum += term;
        coeff *= (double) i / (i + 1);
        hpower *= 0.25;
        term = coeff * hpower / (i + 2);
    }
    printf("Pi is close to %.12g\n", sum * 6);
}
```



9.

```
/*
 * File: circarea.c
 * -----
 * This program estimates the area of a circle with a
 * radius of 1 inch. The program computes the area for
 * the upper right quarter circle and then multiplies
 * the result by 4. The estimate is provided by adding
 * the areas of a series of rectangles of fixed width,
 * where the height of the rectangle is given by the
 * y position on the quarter circle corresponding to the
 * midpoint of the base.
 */

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

/*
 * Constants
 * -----
 * NDivisions -- How many rectangles to use
 * Radius      -- Radius of the circle
 */

#define NDivisions 100
#define Radius      2.0

/* Main program */

main()
{
    double area, dx, x, y;
    int i;

    area = 0;
    dx = Radius / NDivisions;
    x = dx / 2;
    for (i = 0; i < NDivisions; i++) {
        y = sqrt(Radius * Radius - x * x);
        area += dx * y;
        x += dx;
    }
    printf("The area of the circle is %g\n", area);
}
```

10.

```
/*
 * File: eapprox.c
 * -----
 * This program calculates an approximation to the
 * mathematical constant e by summing the series
 *
 *      1      1      1      1      1      1
 *  1 + --- + --- + --- + --- + --- + --- + ...
 *      1!      2!      3!      4!      5!      6!
 *
 */

#include <stdio.h>
#include "genlib.h"

/* Main program */

main()
{
    double sum, term;
    int i;

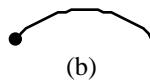
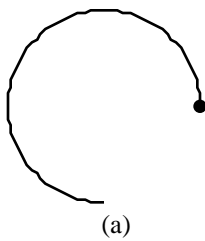
    sum = 0;
    term = 1;
    for (i = 1; sum != sum + term; i++) {
        sum += term;
        term /= i;
    }
    printf("e is close to %.10g\n", sum);
}
```

## Solutions for Chapter 7

### Libraries and Interfaces: A Simple Graphics Library

#### Review questions

1. False. Interfaces will be a continuing theme throughout the book.
2. An *interface* is the boundary between the implementation of a library and programs which use that library. Information passes across that boundary whenever functions in the library are called. A *package* is the `.h` file that serves as an interface together with one or more `.c` files that together provide an implementation. An *abstraction* is the conceptual understanding that an interface represents. The *implementor* is the programmer who codes the functions supplied by an interface. The term *client* is used to refer both to code that makes use of functions supplied by an interface and to the programmer writing that code.
3. The implementor is concerned with the details of how the functions in an interface are implemented. The client does not care about those details but needs to know what each function does and how it is called.
4. Interfaces are represented in C as header files.
5. Header files often consist mostly of comments. In a header file used as an interface, the most important interface entries are the prototypes for the functions the interface exports.
6. A header file used as an interface represents an understanding between the client and the implementor. In most cases, the understanding is best conveyed in English using comments that explain everything the client needs to know, and nothing else.
7. Coordinates in the graphics library are measured in inches from the origin, which is the point in the lower left corner of the graphics window.
8. Absolute coordinates, such as those used by `MovePen`, are specified relative to the origin; relative coordinates, such as those used by `DrawLine`, are specified relative to the previous pen position.
9. The eight functions exported by the graphics library are `InitGraphics`, `MovePen`, `DrawLine`, `DrawArc`, `GetWindowWidth`, `GetWindowHeight`, `GetCurrentX`, and `GetCurrentY`.
10. `InitGraphics();`
11. To change the position of the pen, you call the `MovePen` function.
12. `MovePen(0, 0);`  
`DrawLine(2, 1);`
13. If the third argument to `DrawArc` is negative, the arc is drawn clockwise from its starting point; arcs are ordinarily drawn counterclockwise.
14. In each of the following figures (which are drawn at half-scale), the starting point is shown as a dot.



15. The program produces the following convex-lens shape:



16. You can obtain the coordinates of the center by calling `GetWindowWidth` and `GetWindowHeight` and then dividing each of the returned values by 2.
17. Implementing new procedures provides you with powerful new tools that you can then use over and over again in your programming.
18. If you consider a problem abstractly, you can often design tools that are much more general, which in turn makes them useful in a wider variety of applications.
19. The term *bottom-up implementation* refers to the strategy of implementing the most primitive functions first—precisely the opposite order from that used in top-down design. The main advantage of implementing from the bottom up is that it allows you to test the pieces of the program independently before the entire program is complete.

## Programming exercises

1.

```
/*
 * File: crossbox.c
 * -----
 * This program draws a box plus its diagonals.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"
#include "graphics.h"

/* Function prototypes */

void DrawCrossedBox(double x, double y, double width, double height);
void DrawBox(double x, double y, double width, double height);

/* Test program */

main()
{
    InitGraphics();
    DrawCrossedBox(1.0, 1.0, 1.0, 0.5);
}

/*
 * Function: DrawCrossedBox
 * Usage: DrawCrossedBox(x, y, width, height);
 * -----
 * This function draws a box with its two diagonals.
 */

void DrawCrossedBox(double x, double y, double width, double height)
{
    DrawBox(x, y, width, height);
    MovePen(x, y);
    DrawLine(width, height);
    MovePen(x, y + height);
    DrawLine(width, -height);
}

/* The DrawBox function is included in the text */
```

2.

```
/*
 * File: pyramid.c
 * -----
 * This program draws a pyramid consisting of rows of
 * bricks stacked on top of each other, where the number
 * of bricks in each row is reduced by one for each row.
 */

#include <stdio.h>
#include "genlib.h"
#include "graphics.h"

/* Constants */

#define BrickWidth      (15.0 / 72)
#define BrickHeight     (7.0 / 72)
#define NBricksInBase  16

/* Function prototypes */

void DrawBox(double x, double y, double width, double height);
void DrawGrid(double x, double y, double width, double height,
               int columns, int rows);

/* Main program */

main()
{
    double cx, cy, x, y;
    int nb;

    InitGraphics();
    cx = GetWindowWidth() / 2;
    cy = GetWindowHeight() / 2;
    x = cx - NBricksInBase * BrickWidth / 2;
    y = cy - NBricksInBase * BrickHeight / 2;
    for (nb = NBricksInBase; nb > 0; nb--) {
        DrawGrid(x, y, BrickWidth, BrickHeight, nb, 1);
        x += BrickWidth / 2;
        y += BrickHeight;
    }
}

/* DrawBox and DrawGrid are given in the text */
```

3.

```

/*
 * File: heart.c
 * -----
 * This program draws a heart consisting of two semicircles
 * on top of a square.
 */

#include <stdio.h>
#include <math.h>
#include "genlib.h"
#include "simpio.h"
#include "graphics.h"

/* Function prototypes */

void DrawCenteredHeart(double x, double y, double r);

/* Test program */

main()
{
    double cx, cy, r;

    InitGraphics();
    printf("Radius: ");
    r = GetReal();
    cx = GetWindowWidth() / 2;
    cy = GetWindowHeight() / 2;
    DrawCenteredHeart(cx, cy, r);
}

/*
 * Function: DrawCenteredHeart
 * Usage: DrawCenteredHeart(x, y, r);
 * -----
 * This function draws a heart consisting of two semicircles
 * on top of a square tilted at a 45 degree angle. The center
 * of the square is at (x, y) and the radius of the circles is
 * given by r.
 */

void DrawCenteredHeart(double x, double y, double r)
{
    double root2;

    root2 = sqrt(2);
    MovePen(x - r * root2, y);
    DrawLine(r * root2, -r * root2);
    DrawLine(r * root2, r * root2);
    DrawArc(r, -45, 180);
    DrawArc(r, 45, 180);
}

```

4.

```
/*
 * Function: DrawPeaceSymbol
 * Usage: DrawPeaceSymbol(x, y, r);
 * -----
 * This function draws a peace symbol of radius r centered
 * at (x, y).
 */

void DrawPeaceSymbol(double x, double y, double r)
{
    double root2;

    root2 = sqrt(2);
    DrawCenteredCircle(x, y, r);
    MovePen(x, y - r);
    DrawLine(0, 2 * r);
    MovePen(x, y);
    DrawLine(-r / root2, -r / root2);
    MovePen(x, y);
    DrawLine(r / root2, -r / root2);
}
```



5.

```
/*
 * Function: DrawRoundedBox
 * Usage: DrawRoundedBox(x, y, width, height);
 * -----
 * This function is identical in operation to DrawBox except
 * that the corners of the box are replaced by quarter circles,
 * giving the box a rounded appearance. The radius of the
 * quarter circle is given by the constant CornerRadius, but
 * this value is only used if the box is large enough to include
 * the complete corner. If not, the circle radius is chosen to
 * be half of the smaller of the width and the height.
 */

void DrawRoundedBox(double x, double y,
                   double width, double height)
{
    double r;

    r = MinF(CornerRadius, MinF(width, height) / 2);
    MovePen(x + r, y);
    DrawLine(width - 2 * r, 0);
    DrawArc(r, -90, 90);
    DrawLine(0, height - 2 * r);
    DrawArc(r, 0, 90);
    DrawLine(-(width - 2 * r), 0);
    DrawArc(r, 90, 90);
    DrawLine(0, -(height - 2 * r));
    DrawArc(r, 180, 90);
    MovePen(x, y);
}

/*
 * Function: MinF
 * Usage: min = MinF(x, y);
 * -----
 * This function returns the smaller of the two floating-point
 * values x and y.
 */

double MinF(double x, double y)
{
    if (x < y) {
        return (x);
    } else {
        return (y);
    }
}
```

6.

```
/*
 * File: initials.c
 * -----
 * This program draws the initials ESR. To draw another
 * set of initials, it would be necessary to define new
 * letter procedures.
 */

#include <stdio.h>

#include "genlib.h"
#include "graphics.h"

/*
 * Constants
 * -----
 * LetterHeight    The height of each letter
 * LetterSpacing   The horizontal space between letters
 */

#define LetterHeight 1.0
#define LetterSpacing 0.3

/* Function prototypes */


void LetterE(void);
void LetterS(void);
void LetterR(void);
void AdjustPen(double dx, double dy);

/* Main program */

main()
{
    InitGraphics();
    MovePen(1.0, 0.5);
    LetterE();
    LetterS();
    LetterR();
}

/*
 * Functions: LetterE, LetterS, LetterR
 * -----
 * These functions each draw their indicated letter on
 * the screen, with the lower left corner at the current
 * point. Each function leaves the pen correctly positioned
 * to draw the next letter.
 */

void LetterE(void)
{
    DrawLine(0, LetterHeight);
    DrawLine(LetterHeight * 0.5, 0);
    AdjustPen(-LetterHeight * 0.5, -LetterHeight * 0.5);
    DrawLine(LetterHeight * 0.35, 0);
    AdjustPen(-LetterHeight * 0.35, -LetterHeight * 0.5);
    DrawLine(LetterHeight * 0.5, 0);
    AdjustPen(LetterSpacing, 0);
}
```



```
void LetterS(void)
{
    AdjustPen(LetterHeight * 0.5, LetterHeight * 0.75);
    DrawArc(LetterHeight * 0.25, 0, 270);
    DrawArc(LetterHeight * 0.25, 90, -270);
    AdjustPen(LetterHeight * 0.5 + LetterSpacing,
              -LetterHeight * 0.25);
}

void LetterR(void)
{
    DrawLine(0, LetterHeight);
    DrawLine(LetterHeight * 0.25, 0);
    DrawArc(LetterHeight * 0.25, 90, -180);
    DrawLine(-LetterHeight * 0.25, 0);
    AdjustPen(LetterHeight * 0.15, 0);
    DrawLine(LetterHeight * 0.35, -LetterHeight * 0.5);
    AdjustPen(LetterSpacing, 0);
}

/* The AdjustPen function is given in the text */
```

7.

```

/*
 * File: pumpkin.c
 * -----
 * This program draws a Halloween pumpkin.
 */

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

#include "genlib.h"
#include "graphics.h"

/* Constants */

#define HeadRadius    1.0
#define StemWidth     0.1
#define StemHeight    0.15
#define EyeWidth      0.3
#define EyeHeight     0.2
#define NoseWidth     0.2
#define NoseHeight    0.2
#define NTeethPerRow  7
#define ToothWidth    0.083333
#define ToothHeight   0.15

/* Function prototypes */

void DrawPumpkin(double x, double y);
void DrawStem(double x, double y);
void DrawEye(double x, double y);
void DrawNose(double x, double y);
void DrawMouth(double x, double y);
void DrawBox(double x, double y, double width, double height);
void DrawTriangle(double x, double y, double base, double height);
void DrawCenteredCircle(double x, double y, double r);
void DrawGrid(double x, double y, double width, double height,
              int columns, int rows);

/* Main program */

main()
{
    InitGraphics();
    DrawPumpkin(GetWindowWidth() / 2, GetWindowHeight() / 2);
}

/*
 * Function: DrawPumpkin
 * Usage: DrawPumpkin(x, y);
 * -----
 * This function draws a Halloween pumpkin centered at (x, y).
 * The eyes are positioned halfway to the edge of the circle
 * measured along the diagonal, which means that their x and y
 * offsets must be adjusted by the square root of 2. The
 * distance to each eye along either coordinate axis is
 * given by eyeOffset.
 */

```

```
void DrawPumpkin(double x, double y)
{
    double eyeOffset;

    eyeOffset = HeadRadius / sqrt(2) / 2;
    DrawCenteredCircle(x, y, HeadRadius);
    DrawStem(x, y + HeadRadius);
    DrawEye(x - eyeOffset, y + eyeOffset);
    DrawEye(x + eyeOffset, y + eyeOffset);
    DrawNose(x, y);
    DrawMouth(x, y - HeadRadius / 2);
}

/*
 * Function: DrawStem
 * Usage: DrawStem(x, y);
 * -----
 * This function draws the stem of the pumpkin, which consists
 * of three sides of a rectangle. The (x, y) coordinate is the
 * bottom center of the stem.
 */

void DrawStem(double x, double y)
{
    DrawBox(x - StemWidth / 2, y, StemWidth, StemHeight);
}

/*
 * Function: DrawEye
 * Usage: DrawEye(x, y);
 * -----
 * This function draws an eye centered at (x, y). The current
 * implementation is a downward triangle.
 */

void DrawEye(double x, double y)
{
    DrawTriangle(x - EyeWidth / 2, y + EyeHeight / 2,
                EyeWidth, -EyeHeight);
}

/*
 * Function: DrawNose
 * Usage: DrawNose(x, y);
 * -----
 * This function draws a nose centered at (x, y). The current
 * implementation is an upward triangle.
 */

void DrawNose(double x, double y)
{
    DrawTriangle(x - NoseWidth / 2, y - NoseHeight / 2,
                NoseWidth, NoseHeight);
}
```

```

/*
 * Function: DrawMouth
 * Usage: DrawMouth(x, y);
 * -----
 * This function draws a mouth consisting of two rows of
 * rectangular teeth centered at the point (x, y). The
 * mouth is just a grid exactly like the windows in the
 * house.c example. The only extra work is computing the
 * position of the lower left corner (xc, yc).
 */

void DrawMouth(double x, double y)
{
    double xc, yc;

    xc = x - NTeethPerRow * ToothWidth / 2;
    yc = y - ToothHeight;
    DrawGrid(xc, yc, ToothWidth, ToothHeight, NTeethPerRow, 2);
}

/* The remaining functions are given in the text */

```

8.

```

/*
 * File: usher.c
 * -----
 * This program draws a representation of the house of Usher.
 */

#include <stdio.h>

#include "genlib.h"
#include "graphics.h"

/*
 * Constants
 * -----
 * The following constants control the sizes of the
 * various elements in the display.
 */

#define HouseWidth      1.5
#define HouseHeight    2.0
#define HouseArch       1.0

#define TowerWidth      0.4
#define TowerHeight    2.3
#define TowerArch       0.6

#define DoorWidth       0.3
#define DoorHeight      0.5
#define DoorArch        0.25

#define WindowLevel     1.4
#define WindowSize      0.3

```

```
/* Function prototypes */

void DrawHouseOfUsher(double x, double y);
void DrawTower(double x, double y);
void DrawMainHouse(double x, double y);
void DrawDoor(double x, double y);
void DrawWindow(double x, double y);
void DrawPeakedBox(double x, double y,
                   double width, double height, double theight);
void DrawBox(double x, double y, double width, double height);
void DrawTriangle(double x, double y, double base, double height);

/* Main program */

main()
{
    double cx, cy;

    InitGraphics();
    cx = GetWindowWidth() / 2;
    cy = GetWindowHeight() / 2;
    DrawHouseOfUsher(cx - HouseWidth / 2 - TowerWidth,
                    cy - (HouseHeight + HouseArch) / 2);
}

/*
 * Function: DrawHouseOfUsher
 * Usage: DrawHouseOfUsher(x, y);
 * -----
 * This function draws the complete House of Usher diagram
 * with its lower left corner at (x, y).
 */

void DrawHouseOfUsher(double x, double y)
{
    DrawTower(x, y);
    DrawMainHouse(x + TowerWidth, y);
    DrawTower(x + TowerWidth + HouseWidth, y);
}

/*
 * Function: DrawTower
 * Usage: DrawTower(x, y);
 * -----
 * This function draws one of the side towers. The lower left
 * corner of the tower is at (x, y).
 */

void DrawTower(double x, double y)
{
    DrawPeakedBox(x, y, TowerWidth, TowerHeight, TowerArch);
}
```

```

/*
 * Function: DrawMainHouse
 * Usage: DrawMainHouse(x, y);
 * -----
 * This function draws the main part of the house, including
 * the door and windows. The lower left corner is at (x, y).
 */

void DrawMainHouse(double x, double y)
{
    double xleft, xright;

    DrawPeakedBox(x, y, HouseWidth, HouseHeight, HouseArch);
    DrawDoor(x + (HouseWidth - DoorWidth) / 2, y);
    xleft = x + HouseWidth * 0.25;
    xright = x + HouseWidth * 0.75;
    DrawWindow(xleft - WindowSize / 2, y + WindowLevel);
    DrawWindow(xright - WindowSize / 2, y + WindowLevel);
}

/*
 * Function: DrawDoor
 * Usage: DrawDoor(x, y);
 * -----
 * This function draws the door at (x, y).
 */

void DrawDoor(double x, double y)
{
    DrawPeakedBox(x, y, DoorWidth, DoorHeight, DoorArch);
}

/*
 * Function: DrawWindow
 * Usage: DrawWindow(x, y);
 * -----
 * This function draws a single window with the lower left
 * corner at (x, y).
 */

void DrawWindow(double x, double y)
{
    DrawBox(x, y, WindowSize, WindowSize);
}

/*
 * Function: DrawPeakedBox
 * Usage: DrawPeakedBox(x, y, width, height, theight);
 * -----
 * This function draws a rectangle with a triangular top. The
 * arguments are as in DrawBox, with an additional theight
 * parameter indicating the height of the triangle. This
 * function is a common element in several parts of the picture.
 */

void DrawPeakedBox(double x, double y,
                   double width, double height, double theight)
{
    DrawBox(x, y, width, height);
    DrawTriangle(x, y + height, width, theight);
}

/* DrawBox and DrawTriangle are given in the text. */

```



9.

```

/*
 * File: lincoln.c
 * -----
 * This program draws a simplified picture of the Lincoln
 * Memorial in Washington DC.
 */

#include <stdio.h>
#include "genlib.h"
#include "graphics.h"

/*
 * Constants
 * -----
 * The names of these constants should be sufficient to
 * indicate their function if you look at the output display.
 */

#define MemorialWidth      4.0
#define PedestalHeight    0.3

#define NumberOfColumns   12
#define ColumnWidth       0.15
#define ColumnHeight      1.0
#define ColumnCircleRadius 0.05

#define LowerRoofHeight    0.3
#define UpperRoofWidth    3.5
#define UpperRoofHeight   0.3

#define StatueWidth       0.1
#define StatueHeight      0.2

/* Function prototypes */

void DrawMemorial(double x, double y);
void DrawColumns(double x, double y);
void DrawStatue(double x, double y);
void DrawBox(double x, double y, double width, double height);
void DrawCenteredCircle(double x, double y, double r);

/* Main program */

main()
{
    double cx, cy;
    double totalHeight;

    InitGraphics();
    cx = GetWindowWidth() / 2;
    cy = GetWindowHeight() / 2;
    totalHeight = PedestalHeight + ColumnHeight +
                  LowerRoofHeight + UpperRoofHeight;
    DrawMemorial(cx - MemorialWidth / 2, cy - totalHeight / 2);
}

```

```

/*
 * Function: DrawMemorial
 * Usage: DrawMemorial(x, y);
 * -----
 * This function draws a picture of the Lincoln Memorial. The
 * drawing's lower left corner is point (x,y).
 *
 * Implementation notes:
 * This function draws the pedestal, a row of columns with
 * circles above the column, an upper roof and a lower roof.
 * yLower is the y coordinate of the bottom of the LowerRoof.
 * xUpper and yUpper are the x and y coordinates of the lower
 * left corner of the upper roof
 */

void DrawMemorial(double x, double y)
{
    double yLower, xUpper, yUpper;

    DrawBox(x, y, MemorialWidth, PedestalHeight);
    DrawColumns(x, y + PedestalHeight);
    DrawStatue(x + MemorialWidth / 2 - StatueWidth / 2,
               y + PedestalHeight);
    yLower = y + PedestalHeight + ColumnHeight;
    DrawBox(x, yLower, MemorialWidth, LowerRoofHeight);
    xUpper = x + (MemorialWidth - UpperRoofWidth) / 2;
    yUpper = yLower + LowerRoofHeight;
    DrawBox(xUpper, yUpper, UpperRoofWidth, UpperRoofHeight);
}

/*
 * Function DrawColumns
 * Usage: DrawColumns(x, y);
 * -----
 * This function draws the row of columns, using the point (x,y)
 * as the lower left corner of the row. The size and number of
 * the columns are defined by implementation parameters: there
 * are NumberOfColumns columns, with dimensions ColumnHeight and
 * ColumnWidth. Just above each column, there is a circle
 * centered in the lower roof as shown in the text.
 *
 * Implementation note:
 * The variable delta is the horizontal space between the
 * start of each column. It is computed by noting that the
 * columns must all fit inside the width of the memorial.
 * The variables cx and cy specify the center of the circle.
 */

void DrawColumns(double x, double y)
{
    int i;
    double cx, cy, delta;

    delta = (MemorialWidth - ColumnWidth) /
            (NumberOfColumns - 1);
    for (i = 0; i < NumberOfColumns; i++) {
        DrawBox(x + i * delta, y, ColumnWidth, ColumnHeight);
        cx = x + i * delta + ColumnWidth / 2;
        cy = y + ColumnHeight + LowerRoofHeight / 2;
        DrawCenteredCircle(cx, cy, ColumnCircleRadius);
    }
}

```

```

/*
 * Function: DrawStatue
 * Usage: DrawStatue(x, y)
 * -----
 * This function draws a statue at (x, y). The statue is a box
 * with the dimensions StatueWidth and StatueHeight and a circle
 * on top whose diameter is the width of the body.
 */

void DrawStatue(double x, double y)
{
    double cx, cy;

    DrawBox(x, y, StatueWidth, StatueHeight);
    cx = x + (StatueWidth / 2);
    cy = y + StatueHeight + StatueWidth / 2;
    DrawCenteredCircle(cx, cy, StatueWidth / 2);
}

/* DrawBox and DrawCenteredCircle are given in the text */

```

10.

```

/*
 * File: shadebox.c
 * -----
 * This program draws a shaded box using diagonal lines
 * for shading.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"
#include "graphics.h"

/*
 * Constant: LineSeparation
 * -----
 * This constant specifies the distance between the
 * diagonal lines used to provide shading, measured
 * along either of the cartesian axes.
 */

#define ShadingSeparation 5

/* Function prototypes */

void DrawShadedBox(double x, double y,
                  double width, double height, int sep);
void DrawBox(double x, double y, double width, double height);
double MinF(double x, double y);

```

```

/* Test program */

main()
{
    double cx, cy, width, height;

    InitGraphics();
    DrawShadedBox(1.0, 1.0, 2.0, 0.75, 5);
}

/*
 * Function: DrawShadedBox
 * Usage: DrawShadedBox(x, y, width, height, sep);
 * -----
 * This function draws a box like DrawBox but shades it using
 * diagonal ruled lines. The separation of the lines (measured
 * along the edges of the box rather than the diagonals) is
 * specified by the parameter sep, which is measured in points.
 */

void DrawShadedBox(double x, double y,
                  double width, double height, int sep)
{
    double sx, sy, dx, inchSep, limit;

    DrawBox(x, y, width, height);
    inchSep = sep / 72.0;
    sy = y;
    limit = x + width;
    for (sx = x; sx < limit; sx += inchSep) {
        dx = MinF(height, limit - sx);
        MovePen(sx, sy);
        DrawLine(dx, dx);
    }
    sx = x;
    limit = y + height;
    for (sy = y + inchSep; sy < limit; sy += inchSep) {
        dx = MinF(width, limit - sy);
        MovePen(sx, sy);
        DrawLine(dx, dx);
    }
}

/* The DrawBox function is given in the text. */

/*
 * Function: MinF
 * Usage: min = MinF(x, y);
 * -----
 * This function returns the smaller of the two floating-point
 * values x and y.
 */

double MinF(double x, double y)
{
    if (x < y) {
        return (x);
    } else {
        return (y);
    }
}

```

11.

```

/*
 * File: checker.c
 * -----
 * This program draws a checkerboard centered in the window.
 */

#include <stdio.h>
#include "genlib.h"
#include "graphics.h"

/* Parameters */

#define ShadingSeparation 3
#define SquareSize 0.25

/* Function prototypes */

void DrawCheckerboard(double x, double y);
void DrawShadedBox(double x, double y,
                   double width, double height, int sep);
void DrawBox(double x, double y, double width, double height);
double MinF(double x, double y);

/* Main program */

main()
{
    double cx, cy;

    InitGraphics();
    cx = GetWindowWidth() / 2;
    cy = GetWindowHeight() / 2;
    DrawCheckerboard(cx - 4 * SquareSize, cy - 4 * SquareSize);
}

/*
 * Function: DrawCheckerboard
 * Usage: DrawCheckerboard(x, y);
 * -----
 * This function draws a standard 8 x 8 checkerboard with its
 * lower left corner at (x, y).
 */

void DrawCheckerboard(double x, double y)
{
    int i, j;
    double sx, sy;

    for (i = 0; i < 8; i++) {
        sx = x + i * SquareSize;
        for (j = 0; j < 8; j++) {
            sy = y + j * SquareSize;
            if ((i + j) % 2 == 0) {
                DrawShadedBox(sx, sy, SquareSize, SquareSize,
                              ShadingSeparation);
            } else {
                DrawBox(sx, sy, SquareSize, SquareSize);
            }
        }
    }
}

/* The remaining functions are the same as in Exercise 10 */

```

## Solutions for Chapter 8

### Designing Interfaces: A Random Number Library

#### Review questions

1. False. The hardest thing about writing an interface is finding the best design.
2. A well-designed interface must be unified, simple, sufficient, general, and stable.
3. An abstraction boundary is conceptually the same thing as an interface. The interface acts to separate the world of the client from that of the implementation and therefore acts as a boundary between different abstraction domains.
4. If an interface is stable, both the implementor and the client of that interface are free to change the code under their control without affecting that of the other. When an interface change occurs, all clients of the interface must change their code.
5. A pseudo-random number is a numeric value that appears to be random in a statistical sense even though it is generated by an algorithmic process.
6. On most computers, the value of `RAND_MAX` is the largest positive integer.
7. The four steps necessary to convert the result of `rand` into an integer are normalization, scaling, truncation, and translation.
8. `RandomInteger(1, 100)`
9. The function `RandomInteger` works correctly with negative arguments. The possible results of calling `RandomInteger(-5, 5)` are therefore -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, and 5.
10. No. The statement

```
d1 = d2 = RandomInteger(1, 6);
```

always assigns `d1` and `d2` the same value because the value assigned to `d1` is always the result of the embedded assignment to `d2`.

11. Besides comments, the three most common interface entries are function prototypes, constant definitions, and type definitions.
12. The interface boilerplate for the file `magic.h` would look like this:

```
#ifndef _magic_h
#define _magic_h
. . . contents of the interface . . .
#endif
```

These lines ensure that the compiler processes the contents of the interface only once, even if it is included several times.

13. True. Unless you take special steps to change the random number seed, the `rand` function generates the same sequence of pseudo-random numbers every time.
14. Each returned value of `rand` is computed from the previous value, which is stored internally between calls to the random number generator. This saved value is called the *seed*. If you set the seed to some unpredictable value (usually obtained from the system clock), the pseudo-random number sequence will no longer be the same for each run of the program.

15. When debugging a program involving random numbers, it is best to eliminate any calls to **Randomize**. Programs without a call to **Randomize** will behave deterministically, which means that the conditions of a previous failure can easily be reestablished.
16. The final version of the **random.h** interface contains the following functions:

<b>Randomize</b>	Makes the random sequence unpredictable
<b>RandomInteger</b>	Simulates most random processes with discrete outcomes
<b>RandomReal</b>	Simulates continuous random processes
<b>RandomChance</b>	Simulates the occurrence of a probabilistic event

### Programming exercises

1. The **randtest.c** program is given in the text.

2.

```

/*
 * File: randeven.c
 * -----
 * This program displays a random even number.
 */

#include <stdio.h>
#include "genlib.h"
#include "random.h"

/*
 * Constants
 * -----
 * MinRange -- Minimum number in range
 * MaxRange -- Maximum number in range
 */

#define MinRange 2
#define MaxRange 100

/*
 * Main program
 * -----
 * The general strategy here is to divide each of the limits by
 * two, generate a random integer in that range, and then multiply
 * the result by 2. The only part of the program that might cause
 * confusion is the fact that 1 added to MinRange before the
 * division. This step is necessary only if MinRange might be odd.
 * Allowing C to truncate the value would result in an extra
 * possibility.
 */

main()
{
    Randomize();
    printf("%d\n", 2 * RandomInteger((MinRange + 1) / 2, MaxRange / 2));
}

```

3.

```
/*
 * File: rphone.c
 * -----
 * This program displays a random phone number.
 */

#include <stdio.h>
#include "genlib.h"
#include "random.h"

/* Main program */

main()
{
    Randomize();
    printf("%d", RandomInteger(2, 9));
    printf("%d", RandomInteger(2, 9));
    printf("%d", RandomInteger(0, 9));
    printf("-%04d\n", RandomInteger(0, 9999));
}
```



4.

```
/*
 * File: pickcard.c
 * -----
 * This program picks a random card out of a standard deck of
 * 52 cards and then displays the card value using its full
 * name, such as "Ace of Spades" or "10 of Diamonds".
 */

#include <stdio.h>
#include "genlib.h"
#include "random.h"

/* Function prototypes */

void PrintCard(int rank, int suit);
void PrintCardRank(int rank);
void PrintCardSuit(int suit);

/* Main program */

main()
{
    int rank, suit;
    string flip;

    Randomize();
    rank = RandomInteger(1, 13);
    suit = RandomInteger(1, 4);
    PrintCard(rank, suit);
    printf("\n");
}

/*
 * Function: PrintCard
 * Usage: PrintCard(rank, suit);
 * -----
 * This function displays the full name of a card given
 * numeric values of rank and suit. The output does not
 * include a newline character so that callers can display
 * more than one card name on the same line.
 */

void PrintCard(int rank, int suit)
{
    PrintCardRank(rank);
    printf(" of ");
    PrintCardSuit(suit);
}
```

```
/*
 * Function: PrintCardRank
 * Usage: PrintCardRank(rank);
 * -----
 * This function displays the full name corresponding to numeric
 * value of rank. For cards in the range 2-10, the value is simply
 * printed as a number. For the rank values, 1, 11, 12, and
 * 13, the rank appears as Ace, Jack, Queen, or King.
 */

void PrintCardRank(int rank)
{
    switch (rank) {
        case 1: printf("Ace"); break;
        case 11: printf("Jack"); break;
        case 12: printf("Queen"); break;
        case 13: printf("King"); break;
        default: printf("%d", rank); break;
    }
}

/*
 * Function: PrintCardSuit
 * Usage: PrintCardSuit(suit);
 * -----
 * This function displays the name corresponding to the integer
 * value suit, which will be one of Clubs, Diamonds, Hearts, Spades.
 */

void PrintCardSuit(int suit)
{
    switch (suit) {
        case 1: printf("Clubs"); break;
        case 2: printf("Diamonds"); break;
        case 3: printf("Hearts"); break;
        case 4: printf("Spades"); break;
        default: Error("Illegal suit number %d", suit);
    }
}
```

5.

```
/*
 * File: heads.c
 * -----
 * This program performs a random experiment consisting of flipping
 * a coin repeatedly until ConsecutiveHeads heads appear in a row.
 */

#include <stdio.h>
#include "genlib.h"
#include "strlib.h"
#include "random.h"

/*
 * Constants
 * -----
 * ConsecutiveHeads -- number of consecutive heads desired
 */

#define ConsecutiveHeads 3

/* Function prototypes */

string FlipCoin(void);

/* Implementation */

main()
{
    int nFlips, nHeads;
    string flip;

    Randomize();
    nFlips = 0;
    nHeads = 0;
    while (nHeads < ConsecutiveHeads) {
        flip = FlipCoin();
        printf("%s\n", flip);
        nFlips++;
        if (StringEqual(flip, "heads")) {
            nHeads++;
        } else {
            nHeads = 0;
        }
    }
    printf("It took %d flips to get heads %d consecutive times.\n",
           nFlips, ConsecutiveHeads);
}

/*
 * Function: FlipCoin
 * Usage: str = FlipCoin();
 * -----
 * This function simulates a coin flip by returning
 * either "heads" or "tails" with equal probability.
 */

string FlipCoin(void)
{
    if (RandomChance(0.50)) {
        return ("heads");
    } else {
        return ("tails");
    }
}
```

6.

```
/*
 * File: randompi.c
 * -----
 * This program computes an approximation to pi by simulating
 * a dart board, as described in the text. The general technique
 * is called Monte Carlo integration.
 */

#include <stdio.h>
#include "genlib.h"
#include "random.h"

/*
 * Constants
 * -----
 * NTrials -- number of "darts" to throw.
 */

#define NTrials 10000

/* Main program */

main()
{
    int i, inside;
    double x, y, pi;

    inside = 0;
    for (i = 0; i < NTrials; i++) {
        x = RandomReal(-1.0, 1.0);
        y = RandomReal(-1.0, 1.0);
        if (x * x + y * y < 1.0) inside++;
    }
    pi = 4.0 * inside / NTrials;
    printf("Pi is approximately %g\n", pi);
}
```

7.

```
/*
 * File: halflife.c
 * -----
 * This program simulates radioactive decay.
 */

#include <stdio.h>
#include "genlib.h"
#include "random.h"

/*
 * Constants
 * -----
 * InitialAtoms      -- Initial number of radioactive atoms
 * DecayProbability  -- Chance than an atom will decay in a year
 */

#define InitialAtoms      10000
#define DecayProbability  0.5

/* Main program */

main()
{
    int i, nAtoms, nDecay, year;

    printf("Year      Atoms left\n");
    printf("----      -\n");
    nAtoms = InitialAtoms;
    for (year = 0; nAtoms > 0; year++) {
        printf("%3d      %6d\n", year, nAtoms);
        nDecay = 0;
        for (i = 0; i < nAtoms; i++) {
            if (RandomChance(DecayProbability)) nDecay++;
        }
        nAtoms -= nDecay;
    }
    printf("%3d      %6d\n", year, nAtoms);
}
```

8. The `mathquiz.c` program implements the solution to both exercise 8 and 9.

```
/*
 * File: mathquiz.c
 * -----
 * This program acts as a simple teaching tool for
 * elementary school arithmetic. The program asks the
 * student a question involving either addition or subtraction,
 * using random numbers to choose the type of problem and
 * the two numbers involved. If the student gives the correct
 * response, the program goes on and asks another question.
 * The student is given a specified number of chances (NChances)
 * to solve the problem. If the correct answer is not given
 * within that time, the program shows the correct answer and
 * continues.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"
#include "random.h"

/*
 * Constants
 * -----
 * NQuestions -- Number of questions the student is asked
 * NChances -- Number of chances allowed per question
 * MaxNumber -- Largest number allowed in problems or answers
 */

#define NQuestions 5
#define NChances 3
#define MaxNumber 20

/* Function prototypes */

void AskOneQuestion(void);
int GenerateAndPrintQuestion(void);
void CongratulateStudent(int answer);

/* Main program */

main()
{
    int i;

    printf("Welcome to Math Quiz!\n");
    Randomize();
    for (i = 0; i < NQuestions; i++) {
        AskOneQuestion();
    }
}
```

```
/*
 * Function: AskOneQuestion
 * Usage: AskOneQuestion();
 * -----
 * This function generates and displays a question and then
 * handles the student's response. The function returns when
 * the student supplies the right answer or has run out of
 * chances.
 */


void AskOneQuestion(void)
{
    int chancesLeft, correctAnswer, studentAnswer;

    correctAnswer = GenerateAndPrintQuestion();
    studentAnswer = GetInteger();
    chancesLeft = NChances - 1;
    while (chancesLeft > 0 && studentAnswer != correctAnswer) {
        printf("That's incorrect. Try a different answer: ");
        studentAnswer = GetInteger();
        chancesLeft--;
    }
    if (studentAnswer == correctAnswer) {
        CongratulateStudent(correctAnswer);
    } else {
        printf("No, the answer is %d.\n", correctAnswer);
    }
}

/*
 * Function: GenerateAndPrintQuestion
 * Usage: answer = GenerateAndPrintQuestion();
 * -----
 * This function generates and prints an arithmetic question.
 * The function returns the correct answer. Note that the
 * implementation must be careful that the answer lies in
 * the allowable range and not just the numbers in the
 * original question.
 */

int GenerateAndPrintQuestion(void)
{
    int answer, n1, n2;
    string operator;

    if (RandomChance(0.5)) {
        n1 = RandomInteger(1, MaxNumber - 1);
        n2 = RandomInteger(1, MaxNumber - n1);
        answer = n1 + n2;
        operator = "+";
    } else {
        n1 = RandomInteger(2, MaxNumber);
        n2 = RandomInteger(1, n1 - 1);
        answer = n1 - n2;
        operator = "-";
    }
    printf("What is %d %s %d? ", n1, operator, n2);
    return (answer);
}
```



```
/*
 * Function: CongratulateStudent
 * Usage: CongratulateStudent(answer);
 * -----
 * This function generates a variety of different messages
 * indicating that the student has given the right answer.
 * The argument is provided so that the messages can
 * include the correct answer.
 */

void CongratulateStudent(int answer)
{
    switch (RandomInteger(1, 4)) {
        case 1:
            printf("That's the answer!\n");
            break;
        case 2:
            printf("Correct!\n");
            break;
        case 3:
            printf("Congratulations, that's the answer.\n");
            break;
        case 4:
            printf("You got it. The answer is %d.\n", answer);
            break;
    }
}
```

9. See solution to exercise 8.



10.

```
/*
 * File: rcircles.c
 * -----
 * This program draws random circles in the graphics window.
 */

#include <stdio.h>
#include <stdlib.h>
#include "genlib.h"
#include "random.h"
#include "graphics.h"

/*
 * Constants
 * -----
 * NCircles -- Number of circles
 * MinRadius -- Minimum radius
 * MaxRadius -- Maximum radius
 */

#define NCircles 10
#define MinRadius 0.05
#define MaxRadius 0.50

/* Function prototypes */

void DrawCenteredCircle(double x, double y, double r);

/* Main program */

main()
{
    int i;
    double r, x, y, width, height;

    InitGraphics();
    Randomize();
    width = GetWindowWidth();
    height = GetWindowHeight();
    for (i = 0; i < NCircles; i++) {
        r = RandomReal(MinRadius, MaxRadius);
        x = RandomReal(r, width - r);
        y = RandomReal(r, height - r);
        DrawCenteredCircle(x, y, r);
    }
}

/* The DrawCenteredCircle function is given in the text. */
```

```

11. /*
   * File: rwalk.c
   * -----
   * This program simulates a random walk across the screen. The
   * metaphor used in the descriptions is that of a grid of streets
   * in a city, marked off into streets and avenues, with streets
   * going horizontally and avenues going vertically.
   */

#include <stdio.h>
#include <stdlib.h>
#include "genlib.h"
#include "random.h"
#include "graphics.h"

/*
 * Constants
 * -----
 * NStreets    -- Number of streets
 * NAvenues    -- Number of avenues
 * Margin      -- Space at left and bottom of screen
 * BlockSize   -- The length of a block in inches
 */

#define NStreets    10
#define NAvenues    16
#define Margin      (5.0 / 72)
#define BlockSize   (20.0 / 72)

/* Function prototypes */

bool UseAvenue(int avenue, int street);

/*
 * Main program
 * -----
 * This program simulates a random walk on a network of streets.
 * The idea is that someone starts at the northeast corner of a city
 * and walks to the southwest corner by a random route. At each
 * intersection, the person randomly chooses to walk south along
 * an avenue or west along a street. Along the boundary, the direction
 * of motion is chosen so that the person always stays within the map.
 */

main()
{
    int avenue, street;

    InitGraphics();
    Randomize();
    avenue = NAvenues - 1;
    street = NStreets - 1;
    MovePen(Margin + avenue * BlockSize, Margin + street * BlockSize);
    while (avenue > 0 || street > 0) {
        if (UseAvenue(avenue, street)) {
            DrawLine(0, -BlockSize);
            street--;
        } else {
            DrawLine(-BlockSize, 0);
            avenue--;
        }
    }
}

```

```

/*
 * Function: UseAvenue
 * Usage: if (UseAvenue(avenue, street)) . . .
 * -----
 * This function chooses whether to travel by an avenue or a
 * street. If UseAvenue returns TRUE, the next step is taken
 * along an avenue; if UseAvenue returns FALSE, the next step
 * is along a street. In most cases, the result is chosen
 * randomly. If, however, the current position is along a
 * boundary, the direction is forced.
 */

bool UseAvenue(int avenue, int street)
{
    if (avenue == 0) return (TRUE);
    if (street == 0) return (FALSE);
    return (RandomChance(0.50));
}

```

12.

```

/*
 * File: slots.c
 * -----
 * This program simulates the operation of a slot machine.
 */

#include <stdio.h>

#include "genlib.h"
#include "simpio.h"
#include "strlib.h"
#include "random.h"

/*
 * Constants: CHERRY, LEMON, ORANGE, PLUM, BELL, BAR
 * -----
 * These constants indicate the symbols present on each wheel.
 */

#define CHERRY    1
#define LEMON     2
#define ORANGE    3
#define PLUM      4
#define BELL      5
#define BAR       6

/*
 * Constant: InitialCash
 * -----
 * This constant specifies the number of dollars the player
 * has initially.
 */

#define InitialCash 50

```

```
/* Function prototypes */

void GiveInstructions(void);
int PlaySlotMachine(void);
void ShowPayoff(int w1, int w2, int w3);
void DisplayWheels(int w1, int w2, int w3);
string WheelSymbol(int wheel);
int ComputePayoff(int w1, int w2, int w3);
bool WheelSymbolsMatch(int w1, int w2, int w3);
int PayoffForSymbol(int wheel);
bool GetYesOrNo(string prompt);

/*
 * Main program
 * -----
 * The main program gives instructions and then enters a loop
 * that plays multiple games. The main program is responsible
 * for the user interaction and keeping track of the user's cash.
 */

main()
{
    int cash;

    Randomize();
    if (GetYesOrNo("Would you like instructions? ")) {
        GiveInstructions();
    }
    cash = InitialCash;
    while (cash > 0) {
        printf("You have $%d. ", cash);
        if (!GetYesOrNo("Would you like to play? ")) break;
        cash = cash - 1 + PlaySlotMachine();
    }
    if (cash == 0) printf("You ran out of money.\n");
}

/*
 * Function: GiveInstructions
 * Usage: GiveInstructions();
 * -----
 * This function gives instructions on playing the slot machine.
 */

void GiveInstructions(void)
{
    printf("Welcome to the Las Vegas slot machine program!\n");
    printf("Each time you play the slot machine, three wheels\n");
    printf("will spin until it stops on a symbol. You win money\n");
    printf("if any of the following combinations occur (the -\n");
    printf("indicates any symbol):\n");
    ShowPayoff(BAR, BAR, BAR);
    ShowPayoff(BELL, BELL, BELL);
    ShowPayoff(BELL, BELL, BAR);
    ShowPayoff(PLUM, PLUM, PLUM);
    ShowPayoff(PLUM, PLUM, BAR);
    ShowPayoff(ORANGE, ORANGE, ORANGE);
    ShowPayoff(ORANGE, ORANGE, BAR);
    ShowPayoff(CHERRY, CHERRY, CHERRY);
    ShowPayoff(CHERRY, CHERRY, -1);
    ShowPayoff(CHERRY, -1, -1);
}
```

```
/*
 * Function: PlaySlotMachine
 * Usage: payoff = PlaySlotMachine();
 * -----
 * This function plays one round of the slot machine game and
 * returns the payoff.
 */

int PlaySlotMachine(void)
{
    int w1, w2, w3, payoff;

    w1 = RandomInteger(CHERRY, BAR);
    w2 = RandomInteger(CHERRY, BAR);
    w3 = RandomInteger(CHERRY, BAR);
    DisplayWheels(w1, w2, w3);
    payoff = ComputePayoff(w1, w2, w3);
    if (payoff == 0) {
        printf(" -- You lose\n");
    } else {
        printf(" -- You win $%d\n", payoff);
    }
    return (payoff);
}

/*
 * Function: ShowPayoff
 * Usage: ShowPayoff(w1, w2, w3);
 * -----
 * This function displays the payoff for a given combination.
 */

void ShowPayoff(int w1, int w2, int w3)
{
    DisplayWheels(w1, w2, w3);
    printf(" %3d\n", ComputePayoff(w1, w2, w3));
}

/*
 * Function: DisplayWheels
 * Usage: DisplayWheels(w1, w2, w3);
 * -----
 * This function displays the symbols on each of the three wheels.
 * Returning to the next line is the responsibility of the caller.
 */

void DisplayWheels(int w1, int w2, int w3)
{
    printf("%-6s ", WheelSymbol(w1));
    printf("%-6s ", WheelSymbol(w2));
    printf("%-6s", WheelSymbol(w3));
}
```

```
/*
 * Function: WheelSymbol
 * Usage: str = WheelSymbol(wheel);
 * -----
 * This function returns the name of the symbol corresponding
 * to the numeric code given by wheel. If the value is out
 * of range, WheelSymbol returns the string "-".
 */

string WheelSymbol(int wheel)
{
    switch (wheel) {
        case CHERRY: return ("CHERRY");
        case LEMON:  return ("LEMON");
        case ORANGE: return ("ORANGE");
        case PLUM:   return ("PLUM");
        case BELL:   return ("BELL");
        case BAR:    return ("BAR");
        default:     return ("-");
    }
}

/*
 * Function: ComputePayoff
 * Usage: payoff = ComputePayoff(w1, w2, w3);
 * -----
 * Given the values of the three slot machine wheels, this
 * function computes the payoff.
 */

int ComputePayoff(int w1, int w2, int w3)
{
    if (w1 == CHERRY) {
        if (w2 == CHERRY) {
            if (w3 == CHERRY) return (7);
            return (5);
        }
        return (2);
    }
    if (WheelSymbolsMatch(w1, w2, w3)) {
        return (PayoffForSymbol(w1));
    }
    return (0);
}

/*
 * Function: WheelSymbolsMatch
 * Usage: if (WheelSymbolsMatch(w1, w2, w3)) . . .
 * -----
 * This function returns TRUE if the symbols on all three wheels
 * match. A BAR on the third wheel matches any other symbol.
 */

bool WheelSymbolsMatch(int w1, int w2, int w3)
{
    return (w1 == w2 && (w2 == w3 || w3 == BAR));
}
```

```
/*
 * Function: PayoffForSymbol
 * Usage: payoff = PayoffForSymbol(wheel);
 * -----
 * This function returns the payoff for lining up three of a
 * particular symbol.
 */

int PayoffForSymbol(int wheel)
{
    switch (wheel) {
        case LEMON: return (0);
        case ORANGE: return (10);
        case PLUM: return (14);
        case BELL: return (20);
        case BAR: return (250);
        default: Error("This case should not occur");
    }
}

/*
 * Function: GetYesOrNo
 * Usage: if (GetYesOrNo(prompt)) . . .
 * -----
 * This function asks the user the question indicated by prompt
 * and waits for a yes/no response. If the user answers "yes"
 * or "no", the program returns TRUE or FALSE accordingly.
 * If the user gives any other response, the program asks
 * the question again.
 */

bool GetYesOrNo(string prompt)
{
    string answer;

    while (TRUE) {
        printf("%s", prompt);
        answer = GetLine();
        if (StringEqual(answer, "yes")) return (TRUE);
        if (StringEqual(answer, "no")) return (FALSE);
        printf("Please answer yes or no.\n");
    }
}
```

13.

```
/*
 * File: house.c
 * -----
 * This program draws a simple frame house. It is identical
 * to the program from Chapter 7 except that it uses a library
 * of simple graphical figures accessed through the gfigures.h
 * interface.
 */

#include <stdio.h>

#include "genlib.h"
#include "graphics.h"
#include "gfigures.h"

/*
 * Constants
 * -----
 * The following constants control the sizes of the
 * various elements in the display.
 */

#define HouseHeight      2.0
#define HouseWidth      3.0
#define AtticHeight      0.7

#define DoorWidth        0.4
#define DoorHeight       0.7
#define DoorknobRadius   0.03
#define DoorknobInset    0.07

#define PaneHeight       0.25
#define PaneWidth        0.2

#define FirstFloorWindows 0.3
#define SecondFloorWindows 1.25

/* Function prototypes */

void DrawHouse(double x, double y);
void DrawOutline(double x, double y);
void DrawWindows(double x, double y);
void DrawDoor(double x, double y);

/* Main program */

main()
{
    double cx, cy;

    InitGraphics();
    cx = GetWindowWidth() / 2;
    cy = GetWindowHeight() / 2;
    DrawHouse(cx - HouseWidth / 2,
              cy - (HouseHeight + AtticHeight) / 2);
}
```



[illegible]

```
/*
 * Function: DrawWindows
 * Usage: DrawWindows(x, y);
 * -----
 * This function draws all the windows for the house,
 * taking advantage of the fact that the windows are all
 * arranged in two-dimensional grids of equal-sized panes.
 * By calling the function DrawGrid, this implementation
 * can create all of the window structures using a single
 * tool.
 */

void DrawWindows(double x, double y)
{
    double xleft, xright;

    xleft = x + HouseWidth * 0.25;
    xright = x + HouseWidth * 0.75;
    DrawGrid(xleft - PaneWidth * 1.5, y + FirstFloorWindows,
             PaneWidth, PaneHeight, 3, 2);
    DrawGrid(xright - PaneWidth * 1.5, y + FirstFloorWindows,
             PaneWidth, PaneHeight, 3, 2);
    DrawGrid(xleft - PaneWidth, y + SecondFloorWindows,
             PaneWidth, PaneHeight, 2, 2);
    DrawGrid(xright - PaneWidth, y + SecondFloorWindows,
             PaneWidth, PaneHeight, 2, 2);
}
```

```
/*
 * File: gfigures.h
 * -----
 * This interface provides the client with a library of
 * simple graphical figures.
 */

#ifndef _gfigures_h
#define _gfigures_h

/*
 * Function: DrawBox
 * Usage: DrawBox(x, y, width, height)
 * -----
 * This function draws a rectangle of the given width and
 * height with its lower left corner at (x, y).
 */

void DrawBox(double x, double y, double width, double height);

/*
 * Function: DrawTriangle
 * Usage: DrawTriangle(x, y, base, height)
 * -----
 * This function draws an isosceles triangle (i.e., one with
 * two equal sides) with a horizontal base. The coordinate of
 * the left endpoint of the base is (x, y), and the triangle
 * has the indicated base length and height. If height is
 * positive, the triangle points upward. If height is negative,
 * the triangle points downward.
 */

void DrawTriangle(double x, double y, double base, double height);

/*
 * Function: DrawCenteredCircle
 * Usage: DrawCenteredCircle(x, y, r);
 * -----
 * This function draws a circle of radius r with its
 * center at (x, y).
 */

void DrawCenteredCircle(double x, double y, double r);

/*
 * Function: DrawGrid
 * Usage: DrawGrid(x, y, width, height, columns, rows);
 * -----
 * DrawGrid draws rectangles arranged in a two-dimensional
 * grid. As always, (x, y) specifies the lower left corner
 * of the figure.
 */

void DrawGrid(double x, double y, double width, double height,
              int columns, int rows);

#endif
```

```
/*
 * File: gfigures.c
 * -----
 * This file implements the gfigures.h interface.
 */

#include <stdio.h>
#include "genlib.h"
#include "graphics.h"
#include "gfigures.h"

/* Exported functions */

void DrawBox(double x, double y, double width, double height)
{
    MovePen(x, y);
    DrawLine(0, height);
    DrawLine(width, 0);
    DrawLine(0, -height);
    DrawLine(-width, 0);
}

void DrawTriangle(double x, double y, double base, double height)
{
    MovePen(x, y);
    DrawLine(base, 0);
    DrawLine(-base / 2, height);
    DrawLine(-base / 2, -height);
}

void DrawCenteredCircle(double x, double y, double r)
{
    MovePen(x + r, y);
    DrawArc(r, 0, 360);
}

void DrawGrid(double x, double y, double width, double height,
              int columns, int rows)
{
    int i, j;

    for (i = 0; i < columns; i++) {
        for (j = 0; j < rows; j++) {
            DrawBox(x + i * width, y + j * height,
                    width, height);
        }
    }
}
```

14.

```
/*
 * File: calendar.c
 * -----
 * This program is used to generate a calendar for a year
 * entered by the user. It is identical to the calendar.c
 * program from Chapter 5, except that the low-level functions
 * that might be useful for more general applications have
 * been placed in a separate library called caltools.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"
#include "caltools.h"

/* Function prototypes */

void GiveInstructions(void);
int GetYearFromUser(void);
void PrintCalendar(int year);
void PrintCalendarMonth(int month, int year);
void IndentFirstLine(int weekday);

/* Main program */

main()
{
    int year;

    GiveInstructions();
    year = GetYearFromUser();
    PrintCalendar(year);
}

/*
 * Function: GiveInstructions
 * Usage: GiveInstructions();
 * -----
 * This procedure displays instructions to the user.
 */

void GiveInstructions(void)
{
    printf("This program displays a calendar for a full\n");
    printf("year. The year must not be before 1900.\n");
}
```

```
/*
 * Function: GetYearFromUser
 * Usage: year = GetYearFromUser();
 * -----
 * This function reads in a year from the user and returns
 * that value. If the user enters a year before 1900, the
 * function gives the user another chance.
 */

int GetYearFromUser(void)
{
    int year;

    while (TRUE) {
        printf("Which year? ");
        year = GetInteger();
        if (year >= 1900) return (year);
        printf("The year must be at least 1900.\n");
    }
}

/*
 * Function: PrintCalendar
 * Usage: PrintCalendar(year);
 * -----
 * This procedure prints a calendar for an entire year.
 */

void PrintCalendar(int year)
{
    int month;

    for (month = 1; month <= 12; month++) {
        PrintCalendarMonth(month, year);
        printf("\n");
    }
}

/*
 * Function: PrintCalendarMonth
 * Usage: PrintCalendarMonth(month, year);
 * -----
 * This procedure prints a calendar for the given month
 * and year.
 */

void PrintCalendarMonth(int month, int year)
{
    int weekday, nDays, day;

    printf("    %s %d\n", MonthName(month), year);
    printf(" Su Mo Tu We Th Fr Sa\n");
    nDays = MonthDays(month, year);
    weekday = FirstDayOfMonth(month, year);
    IndentFirstLine(weekday);
    for (day = 1; day <= nDays; day++) {
        printf(" %2d", day);
        if (weekday == Saturday) printf("\n");
        weekday = (weekday + 1) % 7;
    }
    if (weekday != Sunday) printf("\n");
}
```

```
/*
 * Function: IndentFirstLine
 * Usage: IndentFirstLine(weekday);
 * -----
 * This procedure indents the first line of the calendar
 * by printing enough blank spaces to get to the position
 * on the line corresponding to weekday.
 */

void IndentFirstLine(int weekday)
{
    int i;

    for (i = 0; i < weekday; i++) {
        printf("  ");
    }
}
```

```
/*
 * File: caltools.h
 * -----
 * This interface provides several functions that are likely
 * to be useful for clients working with calendar dates. The
 * interface exports several constants along with the functions
 * MonthName, MonthDays, FirstDayOfMonth, and IsLeapYear.
 */

#ifndef _caltools_h
#define _caltools_h

#include "genlib.h"      /* Include definitions of bool and string */

/*
 * Constants: Days of the week
 * -----
 * Days of the week are represented by the integers 0-6.
 */

#define Sunday    0
#define Monday    1
#define Tuesday   2
#define Wednesday 3
#define Thursday  4
#define Friday    5
#define Saturday  6
```

```
/* Exported entries */

/*
 * Function: MonthName
 * Usage: name = MonthName(month);
 * -----
 * MonthName converts a numeric month in the range 1-12
 * into the string name for that month.
 */

string MonthName(int month);

/*
 * Function: MonthDays
 * Usage: ndays = MonthName(month, year);
 * -----
 * MonthDays returns the number of days in the indicated
 * month and year.
 */

int MonthDays(int month, int year);

/*
 * Function: FirstDayOfMonth
 * Usage: weekday = FirstDayOfMonth(month, year);
 * -----
 * This function returns the day of the week on which the
 * indicated month begins.
 */

int FirstDayOfMonth(int month, int year);

/*
 * Function: IsLeapYear
 * Usage: if (IsLeapYear(year)) . . .
 * -----
 * This function returns TRUE if year is a leap year.
 */

bool IsLeapYear(int year);

#endif
```



```
/*
 * File: caltools.c
 * -----
 * This file implements the caltools.h interface.
 */

#include <stdio.h>
#include "genlib.h"
#include "caltools.h"

/* Exported entries */

/*
 * Implementation notes: MonthName
 * -----
 * This implementation uses a straightforward switch statement
 * to convert from a numeric month value into its full name.
 */

string MonthName(int month)
{
    switch (month) {
        case 1: return ("January");
        case 2: return ("February");
        case 3: return ("March");
        case 4: return ("April");
        case 5: return ("May");
        case 6: return ("June");
        case 7: return ("July");
        case 8: return ("August");
        case 9: return ("September");
        case 10: return ("October");
        case 11: return ("November");
        case 12: return ("December");
        default: return ("Illegal month");
    }
}

/*
 * Implementation notes: MonthDays
 * -----
 * This function implements the algorithm indicated by the
 * following nursery rhyme:
 *   Thirty days hath September,
 *   April, June, and November,
 *   All the rest have thirty-one,
 *   Except February alone,
 *   Which has twenty-eight, in fine,
 *   And each leap year, twenty-nine.
 */

int MonthDays(int month, int year)
{
    switch (month) {
        case 2:
            if (IsLeapYear(year)) return (29);
            return (28);
        case 4: case 6: case 9: case 11:
            return (30);
        default:
            return (31);
    }
}
```

```
/*
 * Implementation notes: FirstDayOfMonth
 * -----
 * This function simply counts forward from January 1, 1900,
 * which was a Monday.
 */

int FirstDayOfMonth(int month, int year)
{
    int weekday, i;

    weekday = Monday;
    for (i = 1900; i < year; i++) {
        weekday = (weekday + 365) % 7;
        if (IsLeapYear(i)) weekday = (weekday + 1) % 7;
    }
    for (i = 1; i < month; i++) {
        weekday = (weekday + MonthDays(i, year)) % 7;
    }
    return (weekday);
}

/*
 * Implementation notes: IsLeapYear
 * -----
 * This function implements the rule for leap years, which
 * is that leap years come every four years, except that
 * years ending in 00 are leap years only if the year is
 * divisible by 400.
 */

bool IsLeapYear(int year)
{
    return ( ((year % 4 == 0) && (year % 100 != 0))
            || (year % 400 == 0) );
}
```

## Solutions for Chapter 9

### Strings and Characters

#### Review questions

1. An enumeration type is a finite collection of values that are identified by name.
2. One option for representing enumeration types in C is to use `#define` to create integer constants for each of the named values in the enumeration type. The second approach, which is usually preferable, is to use the `enum` facility to create a new type with a specified set of values.
3. `typedef enum { Lose, Draw, Win };`
4. `typedef enum { Lose = -1, Draw, Win };`
5. `North` is 0, `East` is 1, `South` is 2, and `West` is 3.
6. The acronym ASCII stands for the American Standard Code for Information Interchange.
7. The character `'$'` is ASCII code 36, `'@'` is 64, `'\a'` is 7, and `'x'` is 120.
8. The digits are consecutive in all character codings. In ASCII, both the uppercase letters and the lowercase letters form consecutive sequences.
9. The newline character (`'\n'`) is used most often in C programs.
10. The four most useful arithmetic operators on characters are
  - Adding an integer to a character
  - Subtracting an integer from a character
  - Subtracting one character from another
  - Comparing two characters against each other
11. The expression `high - low` subtracts one character from another and is therefore one of the permissible operations. The result is conceptually an integer, so the other operations in the calculation of `k` involve only integer arithmetic. The expression `low + k` consists of adding an integer to a character.
12. Calling `isdigit(5)` returns `FALSE` because the ASCII code 5 does not correspond to a digit. Calling `isdigit('5')` returns `TRUE`.
13. Calling `toupper('5')` returns `'5'`; the `toupper` and `tolower` functions return their argument unchanged if it is not a letter.
14. True. It is legal to use character constants (or any scalar type constants) as `case` expressions in a `switch` statement.
15. A layered abstraction is a hierarchical set of interfaces, each of which presents a new abstraction that is implemented in terms of the preceding one.
16. True.
17. 

a.	<code>StringLength("ABCDE")</code>	5
b.	<code>StringLength("")</code>	0
c.	<code>StringLength("\a")</code>	1
d.	<code>IthChar("ABC", 2)</code>	<code>'C'</code>
e.	<code>Concat("12", ".00")</code>	<code>"12.00"</code>
f.	<code>CharToString('2')</code>	<code>"2"</code>
g.	<code>SubString("ABCDE", 0, 3)</code>	<code>"ABCD"</code>
h.	<code>SubString("ABCDE", 4, 1)</code>	<code>""</code>
i.	<code>SubString("ABCDE", 3, 9)</code>	<code>"DE"</code>
j.	<code>SubString("ABCDE", 3, 3)</code>	<code>"D"</code>

18. To add a new character to the end of a string you need the functions `Concat` and `CharToString`.
19. You cannot use relational operators to compare two string values. You must call a function like `StringCompare` or `StringEqual`.
20. a. `StringEqual("ABCDE", "abcde")`    `FALSE`  
 b. `StringCompare("ABCDE", "ABCDE")`    `0`  
 c. `StringCompare("ABCDE", "ABC")`    *greater than 0*  
 d. `StringCompare("ABCDE", "abcde")`    *less than 0*  
 e. `FindChar('a', "Abracadabra", 0)`    `3`  
 f. `FindString("ra", "Abracadabra", 3)`    `9`  
 g. `FindString("is", "This is a test.", 0)`    `2`  
 h. `FindString("This is a test", "test", 0)`    `-1`
21. a. `ConvertToLowerCase("Catch-22")`    `"catch-22"`  
 b. `StringToInteger(SubString("Catch-22", 5, 7))`    `-22`  
 c. `RealToString(3.140)`    `"3.14"`  
 d. `Concat(IntegerToString(4 / 3), " pi")`    `"1 pi"`

### Programming exercises

- 1.
- ```

/*
 * File: leftfrom.c
 * -----
 * This file is used to test the LeftFrom function.
 */

#include <stdio.h>
#include "genlib.h"

/*
 * Type: directionT
 * -----
 * This type is used to represent the four major directions
 * on a compass.
 */

typedef enum { North, East, South, West } directionT;

/* Function prototypes */

directionT LeftFrom(directionT dir);
string DirectionName(directionT dir);

```

```
/* Main program */

main()
{
    directionT dir;

    printf("This program tests the LeftFrom function.\n");
    for (dir = North; dir <= West; dir++) {
        printf("LeftFrom(%s) = %s\n", DirectionName(dir),
            DirectionName(LeftFrom(dir)));
    }
}

/*
 * Function: LeftFrom
 * Usage: newdir = LeftFrom(dir);
 * -----
 * This function returns the direction that is to the left relative
 * to the argument. This implementation uses modular arithmetic to
 * compute the new direction. Note that using dir + 3 instead of
 * dir - 1 means that no negative values are passed to the % operator.
 */

directionT LeftFrom(directionT dir)
{
    return ((directionT) (((int) dir + 3) % 4));
}

/*
 * Function: DirectionName
 * Usage: str = DirectionName(dir);
 * -----
 * This function returns the string name corresponding to
 * the direction dir.
 */

string DirectionName(directionT dir)
{
    switch (dir) {
        case North: return ("North");
        case East:  return ("East");
        case South: return ("South");
        case West:  return ("West");
        default:    Error("Illegal direction value");
    }
}
```

2.

```
/*
 * File: calendar.c
 * -----
 * This program is used to generate a calendar for a year
 * entered by the user. It is identical to the calendar.c
 * program from Chapter 5, except that the low-level functions
 * that might be useful for more general applications have
 * been placed in a separate library called caltools.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"
#include "caltools.h"

/* Function prototypes */

void GiveInstructions(void);
int GetYearFromUser(void);
void PrintCalendar(int year);
void PrintCalendarMonth(monthT month, int year);
void IndentFirstLine(weekdayT weekday);

/* Main program */

main()
{
    int year;

    GiveInstructions();
    year = GetYearFromUser();
    PrintCalendar(year);
}

/*
 * Function: GiveInstructions
 * Usage: GiveInstructions();
 * -----
 * This procedure displays instructions to the user.
 */

void GiveInstructions(void)
{
    printf("This program displays a calendar for a full\n");
    printf("year. The year must not be before 1900.\n");
}
```

```
/*
 * Function: GetYearFromUser
 * Usage: year = GetYearFromUser();
 * -----
 * This function reads in a year from the user and returns
 * that value. If the user enters a year before 1900, the
 * function gives the user another chance.
 */

int GetYearFromUser(void)
{
    int year;

    while (TRUE) {
        printf("Which year? ");
        year = GetInteger();
        if (year >= 1900) return (year);
        printf("The year must be at least 1900.\n");
    }
}

/*
 * Function: PrintCalendar
 * Usage: PrintCalendar(year);
 * -----
 * This procedure prints a calendar for an entire year.
 */

void PrintCalendar(int year)
{
    monthT month;

    for (month = January; month <= December; month++) {
        PrintCalendarMonth(month, year);
        printf("\n");
    }
}

/*
 * Function: PrintCalendarMonth
 * Usage: PrintCalendarMonth(month, year);
 * -----
 * This procedure prints a calendar for the given month
 * and year.
 */

void PrintCalendarMonth(monthT month, int year)
{
    weekdayT weekday;
    int nDays, day;

    printf("    %s %d\n", MonthName(month), year);
    printf(" Su Mo Tu We Th Fr Sa\n");
    nDays = MonthDays(month, year);
    weekday = FirstDayOfMonth(month, year);
    IndentFirstLine(weekday);
    for (day = 1; day <= nDays; day++) {
        printf(" %2d", day);
        if (weekday == Saturday) printf("\n");
        weekday = (weekday + 1) % 7;
    }
    if (weekday != Sunday) printf("\n");
}
```

```
/*
 * Function: IndentFirstLine
 * Usage: IndentFirstLine(weekday);
 * -----
 * This procedure indents the first line of the calendar
 * by printing enough blank spaces to get to the position
 * on the line corresponding to weekday.
 */

void IndentFirstLine(weekdayT weekday)
{
    int i;

    for (i = 0; i < weekday; i++) {
        printf("  ");
    }
}
```

```
/*
 * File: caltools.h
 * -----
 * This interface provides several functions that are likely
 * to be useful for clients working with calendar dates. The
 * interface exports enumeration types for the month names and
 * the days of the week along with the functions MonthName,
 * MonthDays, FirstDayOfMonth, and IsLeapYear.
 */

#ifndef _caltools_h
#define _caltools_h

#include "genlib.h"      /* Include definitions of bool and string */

/*
 * Type: monthT
 * -----
 * This type represents the months of the year.
 */

typedef enum {
    January = 1, February, March, April, May, June,
    July, August, September, October, November, December
} monthT;

/*
 * Type: weekdayT
 * -----
 * This type represents the days of the week.
 */

typedef enum {
    Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
} weekdayT;
```



```
/* Exported entries */

/*
 * Function: MonthName
 * Usage: name = MonthName(month);
 * -----
 * MonthName converts a month into the string name for that month.
 */

string MonthName(monthT month);

/*
 * Function: MonthDays
 * Usage: ndays = MonthName(month, year);
 * -----
 * MonthDays returns the number of days in the indicated
 * month and year.
 */

int MonthDays(monthT month, int year);

/*
 * Function: FirstDayOfMonth
 * Usage: weekday = FirstDayOfMonth(month, year);
 * -----
 * This function returns the day of the week on which the
 * indicated month begins.
 */

weekdayT FirstDayOfMonth(monthT month, int year);

/*
 * Function: IsLeapYear
 * Usage: if (IsLeapYear(year)) . . .
 * -----
 * This function returns TRUE if year is a leap year.
 */

bool IsLeapYear(int year);

#endif
```

```
/*
 * File: caltools.c
 * -----
 * This file implements the caltools.h interface.
 */

#include <stdio.h>
#include "genlib.h"
#include "caltools.h"

/* Exported entries */

/*
 * Implementation notes: MonthName
 * -----
 * This implementation uses a straightforward switch statement
 * to convert from a numeric month value into its full name.
 */

string MonthName(monthT month)
{
    switch (month) {
        case January:    return ("January");
        case February:   return ("February");
        case March:      return ("March");
        case April:      return ("April");
        case May:        return ("May");
        case June:       return ("June");
        case July:       return ("July");
        case August:     return ("August");
        case September:  return ("September");
        case October:    return ("October");
        case November:   return ("November");
        case December:   return ("December");
        default:         Error("Illegal month");
    }
}

/*
 * Implementation notes: MonthDays
 * -----
 * This function implements the algorithm indicated by the
 * following nursery rhyme:
 *   Thirty days hath September,
 *   April, June, and November,
 *   All the rest have thirty-one,
 *   Except February alone,
 *   Which has twenty-eight, in fine,
 *   And each leap year, twenty-nine.
 */

int MonthDays(monthT month, int year)
{
    switch (month) {
        case February:
            if (IsLeapYear(year)) return (29);
            return (28);
        case September: case April: case June: case November:
            return (30);
        default:
            return (31);
    }
}
```

```
/*
 * Implementation notes: FirstDayOfMonth
 * -----
 * This function simply counts forward from January 1, 1900,
 * which was a Monday.
 */

weekdayT FirstDayOfMonth(monthT month, int year)
{
    weekdayT weekday;
    int i;

    weekday = Monday;
    for (i = 1900; i < year; i++) {
        weekday = (weekday + 365) % 7;
        if (IsLeapYear(i)) weekday = (weekday + 1) % 7;
    }
    for (i = 1; i < month; i++) {
        weekday = (weekday + MonthDays(i, year)) % 7;
    }
    return (weekday);
}

/*
 * Implementation notes: IsLeapYear
 * -----
 * This function implements the rule for leap years, which
 * is that leap years come every four years, except that
 * years ending in 00 are leap years only if the year is
 * divisible by 400.
 */

bool IsLeapYear(int year)
{
    return ( ((year % 4 == 0) && (year % 100 != 0))
            || (year % 400 == 0) );
}
```

3.

```
/*
 * File: iscons.c
 * -----
 * This file tests the implementation of IsConsonant.
 */

#include <stdio.h>
#include <ctype.h>
#include "genlib.h"

/* Function prototypes */

bool IsConsonant(char ch);
bool IsVowel(char ch);

/* Main program */

main()
{
    char ch;

    printf("The English consonants are:\n");
    for (ch = 'A'; ch <= 'Z'; ch++) {
        if (IsConsonant(ch)) printf(" %c", ch);
    }
    printf("\n");
}

/*
 * Function: IsConsonant
 * Usage: if (IsConsonant(ch)) . . .
 * -----
 * This function returns TRUE if ch is a consonant. This
 * implementation uses IsVowel to simplify the coding.
 */

bool IsConsonant(char ch)
{
    return (isalpha(ch) && !IsVowel(ch));
}

/*
 * Function: IsVowel
 * Usage: if (IsVowel(ch)) . . .
 * -----
 * IsVowel returns TRUE if ch is a vowel. This function
 * recognizes vowels in either upper or lower case.
 */

bool IsVowel(char ch)
{
    switch (tolower(ch)) {
        case 'a': case 'e': case 'i': case 'o': case 'u':
            return (TRUE);
        default:
            return (FALSE);
    }
}
```

4.

```
/*
 * File: randword.c
 * -----
 * This file constructs a random word.
 */

#include <stdio.h>
#include "genlib.h"
#include "random.h"
#include "strlib.h"

/*
 * Constants
 * -----
 * NWords      -- Number of words to generate
 * MinLetters  -- Minimum number of letters in word
 * MaxLetters  -- Maximum number of letters in word
 */

#define NWords      5
#define MinLetters  2
#define MaxLetters  8

/* Function prototypes */

string RandomWord(void);
char RandomLetter(void);

/* Main program */

main()
{
    int i;

    printf("This program generates %d random words.\n", NWords);
    for (i = 0; i < NWords; i++) {
        printf("%s\n", RandomWord());
    }
    printf("\n");
}

/*
 * Function: RandomWord
 * Usage: word = RandomWord();
 * -----
 * This function returns a random "word" consisting of
 * a random number of randomly chosen letters.
 */

string RandomWord(void)
{
    string result;
    int i, nLetters;

    result = "";
    nLetters = RandomInteger(MinLetters, MaxLetters);
    for (i = 0; i < nLetters; i++) {
        result = Concat(result, CharToString(RandomLetter()));
    }
    return (result);
}

/* The RandomLetter function is given in the text. */
```

5.

```
/*
 * File: scrabble.c
 * -----
 * This program counts the basic score for a word
 * in Scrabble.  Scrabble is a trademark of Hasbro
 * industries.
 */

#include <stdio.h>
#include <ctype.h>
#include "genlib.h"
#include "simpio.h"
#include "strlib.h"

/* Function prototypes */

int ScrabbleScore(string word);
int LetterValue(char ch);

/* Main program */

main()
{
    string word;
    int score;

    printf("This program tests the ScrabbleScore function.\n");
    printf("Enter words, ending with a blank line.\n");
    while (TRUE) {
        printf("Word: ");
        word = GetLine();
        if (StringEqual(word, "")) break;
        score = ScrabbleScore(word);
        printf("The basic score for '%s' is %d.\n", word, score);
    }
}

/*
 * Function: ScrabbleScore
 * Usage: value = ScrabbleScore(word);
 * -----
 * This function adds up the point values of the letters
 * in word and returns the total score.
 */

int ScrabbleScore(string word)
{
    int i, score;

    score = 0;
    for (i = 0; i < StringLength(word); i++) {
        score += LetterValue(IthChar(word, i));
    }
    return (score);
}
```

```
/*
 * Function: LetterValue
 * Usage: value = LetterValue(ch);
 * -----
 * This function returns the point count for the letter ch if it
 * were played in a Scrabble game. Lowercase letters are assumed
 * to be a blank and therefore score 0 points.
 */

int LetterValue(char ch)
{
    if (islower(ch)) return (0);
    if (!isalpha(ch)) Error("Illegal character %c in word", ch);
    switch (ch) {
        case 'Q': case 'Z':
            return (10);
        case 'J': case 'X':
            return (8);
        case 'K':
            return (5);
        case 'F': case 'H': case 'V': case 'W': case 'Y':
            return (4);
        case 'B': case 'C': case 'M': case 'P':
            return (3);
        case 'D': case 'G':
            return (2);
        default:
            return (1);
    }
}
```

6.

```
/*
 * File: capital.c
 * -----
 * This file implements the Capitalize function.
 */

#include <stdio.h>
#include <ctype.h>
#include "genlib.h"
#include "strlib.h"
#include "simpio.h"

/* Function prototypes */

string Capitalize(string str);

/* Main program */

main()
{
    string str;

    printf("This file tests the Capitalize function.\n");
    while (TRUE) {
        printf("Enter a string: ");
        str = GetLine();
        if (StringEqual(str, "")) break;
        printf("%s\n", Capitalize(str));
    }
}

/*
 * Function: Capitalize
 * Usage: newstr = Capitalize(str);
 * -----
 * This function returns the original string with the first
 * letter converted to upper case and all others converted
 * to lower case. Characters other than letters are not
 * affected.
 */

string Capitalize(string str)
{
    string head, tail;

    head = SubString(str, 0, 0);
    tail = SubString(str, 1, StringLength(str) - 1);
    return (Concat(ConvertToUpperCase(head),
                    ConvertToLowerCase(tail)));
}
```



7.

```
/*
 * File: eqnocase.c
 * -----
 * This program tests the EqualIgnoringCase function.
 */

#include <stdio.h>
#include <ctype.h>
#include "genlib.h"
#include "strlib.h"
#include "simpio.h"

/* Function prototypes */

bool EqualIgnoringCase(string s1, string s2);

/* Main program */

main()
{
    string s1, s2;

    printf("This program tests the EqualIgnoringCase function.\n");
    while (TRUE) {
        printf("s1: ");
        s1 = GetLine();
        if (StringEqual(s1, "")) break;
        printf("s2: ");
        s2 = GetLine();
        if (EqualIgnoringCase(s1, s2)) {
            printf("Equal\n");
        } else {
            printf("Not equal\n");
        }
    }
}

/*
 * Function: EqualIgnoringCase
 * Usage: if (EqualIgnoringCase(s1, s2) . . .
 * -----
 * This function returns TRUE if s1 and s2 are equal, ignoring
 * differences in the case of letters.
 */

bool EqualIgnoringCase(string s1, string s2)
{
    return (StringEqual(ConvertToLowerCase(s1),
                        ConvertToLowerCase(s2)));
}
```

8.

```

/*
 * File: cipher.c
 * -----
 * This program encodes a message by adding a fixed offset to
 * each character within it, cycling from the end of the alphabet
 * back to the beginning.
 */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "genlib.h"
#include "strlib.h"
#include "simpio.h"

/* Function prototypes */

string EncodeString(string str, int key);

/* Main program */

main()
{
    string str;
    int key;

    printf("This program encodes a message using a cyclic cipher.\n");
    printf("Enter the numeric key: ");
    key = GetInteger();
    printf("Enter a message: ");
    str = GetLine();
    printf("Encoded message: %s\n", EncodeString(str, key));
}

/*
 * Function: EncodeString
 * Usage: newstr = EncodeString(str, key);
 * -----
 * This function returns the encoded version of the original string
 * formed by replacing every letter in str by the letter that comes
 * key letters further on in the alphabet. The alphabet is assumed
 * to cycle around so that 'A' comes after 'Z'.
 */

string EncodeString(string str, int key)
{
    string result;
    char ch;
    int i;

    result = "";
    if (key < 0) key = 26 - abs(key) % 26;
    for (i = 0; i < StringLength(str); i++) {
        ch = IthChar(str, i);
        if (islower(ch)) {
            ch = 'a' + ((ch - 'a') + key) % 26;
        } else if (isupper(ch)) {
            ch = 'A' + ((ch - 'A') + key) % 26;
        }
        result = Concat(result, CharToString(ch));
    }
    return (result);
}

```

9.

```
/*
 * File: palin.c
 * -----
 * This file implements and tests the IsPalindrome function.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"
#include "strlib.h"

/* Function prototypes */

bool IsPalindrome(string str);
string ReverseString(string str);

/* Main program */

main()
{
    string str;

    printf("This program checks for palindromes.\n");
    printf("Indicate end of input with a blank line.\n");
    while (TRUE) {
        printf("Enter a string: ");
        str = GetLine();
        if (StringEqual(str, "")) break;
        if (IsPalindrome(str)) {
            printf("'%s' is a palindrome.\n", str);
        } else {
            printf("'%s' is not a palindrome.\n", str);
        }
    }
}

/*
 * Function: IsPalindrome
 * Usage: if (IsPalindrome(str)) . . .
 * -----
 * This function returns TRUE if the string is a palindrome.
 */

bool IsPalindrome(string str)
{
    return (StringEqual(str, ReverseString(str)));
}

/* The ReverseString function is given in the text */
```

10.

```
/*
 * File: palsent.c
 * -----
 * This file tests for sentence palindromes.
 */

#include <stdio.h>
#include <ctype.h>
#include "genlib.h"
#include "simpio.h"
#include "strlib.h"

/* Function prototypes */

bool IsSentencePalindrome(string str);
string RemoveNonAlpha(string str);
string ReverseString(string str);

/* Main program */


main()
{
    string str;

    printf("This program checks for palindromes.\n");
    printf("Indicate the end of the input with a blank line.\n");
    while (TRUE) {
        printf("Enter a string: ");
        str = GetLine();
        if (StringEqual(str, "")) break;
        if (IsSentencePalindrome(str)) {
            printf("That is a palindrome.\n");
        } else {
            printf("That is not a palindrome.\n");
        }
    }
}

/*
 * Function: IsSentencePalindrome
 * Usage: if (IsSentencePalindrome(str)) . . .
 * -----
 * This function returns TRUE if the string is a sentence palindrome,
 * which is defined to be a string in which the letters (ignoring other
 * characters) read the same backwards and forwards.
 */

bool IsSentencePalindrome(string str)
{
    string letters;

    letters = ConvertToLowerCase(RemoveNonAlpha(str));
    return (StringEqual(letters, ReverseString(letters)));
}
```



```
/*
 * Function: RemoveNonAlpha
 * Usage: newstr = RemoveNonAlpha(str);
 * -----
 * This function returns str after removing any nonalphabetic characters.
 */

string RemoveNonAlpha(string str)
{
    string result;
    int i;

    result = "";
    for (i = 0; i < StringLength(str); i++) {
        if (isalpha(IthChar(str, i))) {
            result = Concat(result, CharToString(IthChar(str, i)));
        }
    }
    return (result);
}

/* The ReverseString function is given in the text */
```

```

11.  /*
    * File: datestr.c
    * -----
    * This program tests the DateString function.
    */

#include <stdio.h>
#include <ctype.h>
#include "genlib.h"
#include "simpio.h"
#include "strlib.h"

/* Function prototypes */

string DateString(int day, int month, int year);
string MonthName(int month);

/* Main program */

main()
{
    int day, month, year;

    printf("This program tests the DateString function.\n");
    printf("Enter day of the month (1-31): ");
    day = GetInteger();
    printf("Enter numeric month (1-12): ");
    month = GetInteger();
    printf("Enter year: ");
    year = GetInteger();
    printf("%s\n", DateString(day, month, year));
}

/*
 * Function: DateString
 * Usage: str = DateString(day, month, year);
 * -----
 * This function returns a string representation of the indicated
 * date in the format dd-mmm-yy where dd is the day (one or two
 * digits), mmm is the first three letters of the month name, and
 * yy represents the last two digits of the year. Thus, calling
 * DateString(31, 1, 1994) returns the string "31-Jan-94".
 */

string DateString(int day, int month, int year)
{
    string str;

    str = Concat(IntegerToString(day), "-");
    str = Concat(str, SubString(MonthName(month), 0, 2));
    str = Concat(str, "-");
    str = Concat(str, IntegerToString(year % 100));
    return (str);
}

/* The MonthName function is given in the text */

```

12.

```
/*
 * Function: MyFindString
 * Usage: p = MyFindString(s, text, start);
 * -----
 * This function searches for the string s in the string text,
 * beginning at position start. It returns the first index
 * at which s appears, or -1 if no match is found.
 */

int MyFindString(string s, string text, int start)
{
    int ls, ltext, i;

    ls = StringLength(s);
    ltext = StringLength(text);
    if (start < 0) start = 0;
    for (i = start; i < ltext - ls + 1; i++) {
        if (StringEqual(s, SubString(text, i, i + ls - 1))) {
            return (i);
        }
    }
    return (-1);
}
```

13.

```

/*
 * File: repall.c
 * -----
 * This program tests the function ReplaceAll that replaces
 * all instances of one string with another.
 */

#include <stdio.h>
#include <ctype.h>
#include "genlib.h"
#include "strlib.h"
#include "simpio.h"

/* Function prototypes */

string ReplaceAll(string str, string pattern, string replacement);

/* Main program */

main()
{
    string str, pattern, replacement;

    printf("Program to edit a string by replacing all copies of\n");
    printf("a pattern string by a new replacement string.\n");
    printf("Enter the string to be edited:\n");
    str = GetLine();
    printf("Enter the pattern string: ");
    pattern = GetLine();
    printf("Enter the replacement string: ");
    replacement = GetLine();
    str = ReplaceAll(str, pattern, replacement);
    printf("%s\n", str);
}

/*
 * Function: ReplaceAll
 * Usage: newstr = ReplaceAll(str, pattern, replacement);
 * -----
 * This function searches through the string str and replaces
 * any instances of the pattern string with the replacement
 * string.
 */

string ReplaceAll(string str, string pattern, string replacement)
{
    string result;
    int start, pos, lstr, lpat;

    result = "";
    start = 0;
    lstr = StringLength(str);
    lpat = StringLength(pattern);
    while ((pos = FindString(pattern, str, start)) != -1) {
        result = Concat(result, SubString(str, start, pos - 1));
        result = Concat(result, replacement);
        start = pos + lpat;
    }
    result = Concat(result, SubString(str, start, lstr - 1));
    return (result);
}

```



14.

```
/*
 * File: plural.c
 * -----
 * This program tests the RegularPluralForm function, which uses
 * standard English rules to form the plural of a word.
 */

#include <stdio.h>
#include <ctype.h>
#include "genlib.h"
#include "strlib.h"
#include "simpio.h"

/* Function prototypes */

string RegularPluralForm(string word);
bool IsVowel(char ch);

/* Main program */

main()
{
    string word, plural;

    printf("Program to construct regular plurals.\n");
    while (TRUE) {
        printf("Enter a word: ");
        word = GetLine();
        if (StringEqual(word, "")) break;
        plural = RegularPluralForm(word);
        printf("One %s, two %s.\n", word, plural);
    }
}
```



```
/*
 * Function: RegularPluralForm
 * Usage: plural = RegularPluralForm(word);
 * -----
 * This function returns the regular plural form of word, obtained
 * by applying the following rules:
 *
 * 1. If the word ends in "s", "x", "z", "ch", or "sh", add "es".
 * 2. If the word ends in a "y" preceded by a consonant,
 *    replace the "y" with "ies".
 * 3. In all other cases, add a single "s".
 */

string RegularPluralForm(string str)
{
    int len;

    len = StringLength(str);
    switch (IthChar(str, len - 1)) {
        case 's': case 'x': case 'z':
            return (Concat(str, "es"));
        case 'h':
            if (len > 1) {
                switch (IthChar(str, len - 2)) {
                    case 'c': case 's':
                        return (Concat(str, "es"));
                }
            }
            break;
        case 'y':
            if (len > 1) {
                if (!IsVowel(IthChar(str, len - 2))) {
                    return (Concat(SubString(str, 0, len - 2), "ies"));
                }
            }
            break;
    }
    return (Concat(str, "s"));
}

/* The IsVowel function is given in the text. */
```

## Solutions for Chapter 10

### Modular Development

#### Review questions

1. True.
2. The main module is the one that contains the function `main`.
3. False. In many cases, the modular structure of a program follows from the decomposition strategy you develop during the top-down design process.
4. The term *pseudocode* refers to a mixture of actual C code and English. The English is used to represent parts of the program that have yet to be coded.
5. In many cases, incomplete specifications can be resolved by thinking about the problem and making reasonable choices on your own. If there are several alternative strategies that affect how the final product will behave, it is good practice to discuss those alternatives with the people who will be using the application to see what they want.
6. In scanning a string, a token is a contiguous set of characters that stand together as a meaningful unit. Typically, each word or number in a string represents a separate token, as do the individual punctuation marks.
7. Local and global variables differ in their scope and lifetime. Local variables are available only to the function in which they are declared; global variables can be used by any function in the module that declares it. In terms of lifetime, local variables disappear when the function declaring them returns; global variables retain their values throughout the lifetime of a program.
8. Local variables are declared within the context of a function. Global variable declarations appear at the top level of a module, outside any function definition.
9. See the answer to question 7.
10. Because global variables can be referenced from any function in a module, it is harder to understand how they are being used in the context of a program. In the case of a local variable, all references to that variable are constrained to be within a single function. To understand the behavior of a global variable, you have to look at the entire program. The module-wide scope of a global variable also makes programs harder to debug. If you discover that a global variable is somehow being set to an incorrect value, you have less information about where the error occurred.
11. Using the keyword `static` as part of a variable declaration ensures that the variable can be referenced only within the current module. If you declare a global variable without the `static` keyword, other modules can gain access to that variable, although this technique is never used in the text. The `static` keyword plays a similar role in the declaration of a function. If a function is declared as static, it can only be called from within the module in which it appears.
12. Static initialization is indicated by including an equal sign and a value as part of the declaration of a global variable, and has the effect of storing the indicated value in the variable before program execution even begins. Dynamic initialization is indicated by assignment statements within the program and takes effect only when the program is run. Static initialization is most useful in the following cases:
  - If the value of a variable is constant throughout the lifetime of the program
  - If a variable usually has a particular initial value that only a few clients might want to change

## Programming exercises

1.

```
/*
 * Function: OldStyleAbbreviation
 * Usage: abbrev = OldStyleAbbreviation(word);
 * -----
 * This function returns the old style abbreviation of word, which
 * consists of:
 * 1. The initial consonant substring, up to but not including
 *    the first vowel. If the string begins with a vowel, this
 *    part of the abbreviation consists of just that one vowel
 * 2. The last letter in the word
 * 3. A period
 */

string OldStyleAbbreviation(string str)
{
    int vp, len;
    string head;
    string last;

    vp = FindFirstVowel(str);
    if (vp == -1) Error("No vowels in word");
    if (vp == 0) {
        head = SubString(str, 0, 0);
    } else {
        head = SubString(str, 0, vp - 1);
    }
    len = StringLength(str);
    last = SubString(str, len - 1, len - 1);
    return (Concat(Concat(head, last), "."));
}

/* FindFirstVowel and IsVowel are given in the text. */
```

2. The only parts of the program that need to change are the following functions:

```

/*
 * Function: TranslateLine
 * Usage: TranslateLine(line);
 * -----
 * This function takes a line of text in line and translates
 * the words in the line to Pig Latin, displaying the
 * translation as it goes.
 */

static void TranslateLine(string line)
{
    string word;
    int i, start;
    char ch;

    for (i = 0; i < StringLength(line); i++) {
        ch = IthChar(line, i);
        if (isalnum(ch)) {
            if (IsBeginningOfWord(line, i)) {
                start = i;
            } else if (IsEndOfWord(line, i)) {
                word = SubString(line, start, i);
                word = TranslateWord(word);
                printf("%s", word);
            }
        } else {
            printf("%c", ch);
        }
    }
    printf("\n");
}

/*
 * Functions: IsBeginningOfWord, IsEndOfWord
 * Usage: if (IsBeginningOfWord(line, i)) . . .
 *         if (IsEndOfWord(line, i)) . . .
 * -----
 * These functions return TRUE if the character in position i is
 * at the beginning or end of the word, respectively.
 */

static bool IsBeginningOfWord(string line, int i)
{
    return (isalnum(IthChar(line, i))
            && (i == 0 || !isalnum(IthChar(line, i - 1))));
}

static bool IsEndOfWord(string line, int i)
{
    return (isalnum(IthChar(line, i))
            && (i == StringLength(line) - 1
                || !isalnum(IthChar(line, i + 1))));
}

/* The remaining functions are the same as those in the text. */

```

3. The only required changes are in the following functions:

```

/*
 * Procedure: TranslateWord
 * Usage: word = TranslateWord(word)
 * -----
 * This function translates word from English to Pig Latin,
 * preserving the capitalization of the original word.
 */

string TranslateWord(string word)
{
    bool capitalized;

    capitalized = isupper(IthChar(word, 0));
    word = TranslateLowerCaseWord(ConvertToLowerCase(word));
    if (capitalized) word = Capitalize(word);
    return (word);
}

/*
 * Procedure: TranslateLowerCaseWord
 * Usage: word = TranslateLowerCaseWord(word)
 * -----
 * This is the old TranslateWord function, which is now guaranteed
 * to get a lowercase word.
 */

string TranslateLowerCaseWord(string word)
{
    int vp;
    string head, tail;

    vp = FindFirstVowel(word);
    if (vp == -1) {
        return (word);
    } else if (vp == 0) {
        return (Concat(word, "way"));
    } else {
        head = SubString(word, 0, vp - 1);
        tail = SubString(word, vp, StringLength(word) - 1);
        return (Concat(tail, Concat(head, "ay")));
    }
}

/*
 * Function: Capitalize
 * Usage: newstr = Capitalize(str);
 * -----
 * This function returns the original string with the first letter
 * converted to upper case and all others converted to lower case.
 */

string Capitalize(string str)
{
    string head, tail;

    head = SubString(str, 0, 0);
    tail = SubString(str, 1, StringLength(str) - 1);
    return (Concat(ConvertToUpperCase(head),
                    ConvertToLowerCase(tail)));
}

```

4.

```
/*
 * File: ashtray.c
 * -----
 * This program finds word pairs like trash/ashtray in which
 * the Pig Latin form of the first word is another English word.
 */

#include <stdio.h>
#include <ctype.h>
#include "genlib.h"
#include "strlib.h"
#include "worddict.h"

/* Function prototypes */

static string TranslateWord(string word);
static int FindFirstVowel(string word);
static bool IsVowel(char ch);

/* Main program */

main()
{
    string word, piglatin;

    InitDictionary();
    while (!AtEndOfDictionary()) {
        word = GetNextWord();
        piglatin = TranslateWord(word);
        if (IsEnglishWord(piglatin)) {
            printf("%s - %s\n", word, piglatin);
        }
    }
}

/* TranslateWord, FindFirstVowel, and IsVowel are in the text. */
```



5.

```
/*
 * File: wordcnt.c
 * -----
 * This program counts the number of words in several lines
 * of text, where the input is terminated by a blank line.
 */

#include <stdio.h>
#include <ctype.h>
#include "genlib.h"
#include "strlib.h"
#include "simpio.h"
#include "scanner.h"

/* Function prototypes */

int WordCount(string line);
bool IsLegalWord(string token);

/* Main program */

main()
{
    string line;
    int count;

    printf("This program counts the number of words in a paragraph.\n");
    printf("End the paragraph with a blank line.\n");
    count = 0;
    while (TRUE) {
        line = GetLine();
        if (StringEqual(line, "")) break;
        count += WordCount(line);
    }
    printf("Number of words: %d\n", count);
}

/*
 * Function: WordCount
 * Usage: n = WordCount(line);
 * -----
 * This function returns the number of words in the string line.
 */

int WordCount(string line)
{
    int count;

    InitScanner(line);
    count = 0;
    while (!AtEndOfLine()) {
        if (IsLegalWord(GetNextToken())) count++;
    }
    return (count);
}

/* The IsLegalWord function is given in the text. */
```

6.

```
/*
 * File: longword.c
 * -----
 * This program prints out the longest word in a line.
 */

#include <stdio.h>
#include "genlib.h"
#include "strlib.h"
#include "simpio.h"
#include "scanner.h"

/* Function prototypes */

string LongestWord(string line);

/* Main program */

main()
{
    string line;

    printf("Enter a line:\n");
    line = GetLine();
    printf("The longest word is \"%s\".\n", LongestWord(line));
}

/*
 * Procedure: LongestWord
 * Usage: word = LongestWord(line);
 * -----
 * This function takes a line of text in line and returns the
 * longest word in the line.  If there is more than one word of
 * that length, the function returns the first such word.
 */

string LongestWord(string line)
{
    string token, longest;

    InitScanner(line);
    longest = "";
    while (!AtEndOfLine()) {
        token = GetNextToken();
        if (StringLength(token) > StringLength(longest)) {
            longest = token;
        }
    }
    return (longest);
}
```

7.

```

/*
 * File: revsent.c
 * -----
 * This program constructs a new sentence out of an old one by
 * listing the words in reverse order.
 */

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

#include "genlib.h"
#include "strlib.h"
#include "simpio.h"
#include "scanner.h"

/* Private function prototypes */

string ReverseSentence(string line);
bool IsLegalWord(string token);

/* Main program */

main()
{
    string line;

    printf("Enter a line: ");
    line = GetLine();
    printf("%s\n", ReverseSentence(line));
}

/*
 * Function: ReverseSentence
 * Usage: line = ReverseSentence(line);
 * -----
 * This function takes a line of text and reverses the order
 * of the words. All punctuation marks are discarded, and the
 * words in the new sentence are separated by a single space.
 */

string ReverseSentence(string line)
{
    string token, result;

    InitScanner(line);
    result = "";
    while (!AtEndOfLine()) {
        token = GetNextToken();
        if (IsLegalWord(token)) {
            if (!StringEqual(result, "")) {
                result = Concat(" ", result);
            }
            result = Concat(token, result);
        }
    }
    return (result);
}

/* The IsLegalWord function is given in the text. */

```

8. The following interface and implementation include the solution to both exercise 8 and exercise 9.

```
/*
 * File: scanner.h
 * -----
 * This file is an extended version of the scanner interface
 * described in Chapter 10. The extensions are:
 *
 * o The client can set the package to return uppercase tokens.
 * o The client can set the package to ignore white space.
 *
 * The principal function of the scanner package is to divide a
 * line into individual "tokens," which are either:
 *
 * 1. a string of consecutive letters and digits representing
 *    a word, or
 *
 * 2. a one-character string representing a separator
 *    character, such as a space or punctuation mark.
 *
 * To use this package, you must first call
 *
 *     InitScanner(line);
 *
 * where line is the string (typically a line returned by
 * GetLine) that is to be divided into tokens. To retrieve
 * each token in turn, you call
 *
 *     token = GetNextToken();
 *
 * To detect the end of the line, you can use either of two
 * strategies. The predicate function AtEndOfLine returns
 * TRUE when the last token has been read, so that the loop
 * structure:
 *
 *     while (!AtEndOfLine()) {
 *         token = GetNextToken();
 *         . . . process the token . . .
 *     }
 *
 * serves as an idiom for processing each token on the line.
 * Alternatively, you can also check the result of GetNextToken,
 * which returns the empty string at the end of the line.
 *
 * This version of the interface provides two additional options
 * for the client, as discussed in the exercises for the chapter:
 *
 * (1) The client can request only uppercase tokens by
 *     calling the function ReturnUppercaseTokens(TRUE).
 *     By default, tokens are returned just as they appear
 *     in the input line.
 *
 * (2) The client can request that the scanner ignore
 *     space characters in the input stream by calling
 *     IgnoreSpaces(TRUE). By default, spaces are returned
 *     as separate tokens.
 *
 * Further details for each function are given in the
 * individual descriptions below.
 */
```

```

#ifndef _scanner_h
#define _scanner_h

#include "genlib.h"

/*
 * Function: InitScanner
 * Usage: InitScanner(line);
 * -----
 * This function initializes the scanner and sets it up so that
 * it reads tokens from line. After calling InitScanner, the next
 * call to GetNextToken will return the first token on the line,
 * the next call will return the second token, and so on.
 */

void InitScanner(string line);

/*
 * Function: GetNextToken
 * Usage: word = GetNextToken();
 * -----
 * This function returns the next token on the line. If GetNextToken
 * is called at the end of the input line, it returns the empty string.
 */

string GetNextToken(void);

/*
 * Function: AtEndOfLine
 * Usage: if (AtEndOfLine()) . . .
 * -----
 * This function returns TRUE when the scanner is at the end of the line.
 */

bool AtEndOfLine(void);

/*
 * Function: ReturnUppercaseTokens
 * Usage: ReturnUppercaseTokens(flag);
 * -----
 * This function controls whether or not the scanner returns tokens
 * in their original form or converts them to uppercase. By default,
 * tokens are returned just as they appear in the input line. Calling
 * ReturnUppercaseTokens(TRUE), sets the scanner to return uppercase
 * tokens; ReturnUppercaseTokens(FALSE) restores the default behavior.
 */

void ReturnUppercaseTokens(bool flag);

/*
 * Function: IgnoreSpaces
 * Usage: IgnoreSpaces(flag);
 * -----
 * This function controls whether or not the scanner considers space
 * characters as valid tokens. Ordinarily, space characters are
 * returned by GetNextToken, just like any other separator. If the
 * client calls IgnoreSpaces(TRUE), GetNextToken will skip over spaces
 * before a token; IgnoreSpaces(FALSE) restores the original behavior.
 */

void IgnoreSpaces(bool flag);

#endif

```

```
/*
 * File: scanner.c
 * -----
 * This file implements the extended version of the scanner.h
 * interface. This implementation includes both the option to
 * ignore spaces and the option to convert tokens to uppercase.
 */

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

#include "genlib.h"
#include "strlib.h"
#include "scanner.h"

/*
 * Private variables
 * -----
 * buffer          -- Private copy of string passed to InitScanner
 * buflen          -- Length of the buffer, saved for efficiency
 * cpos            -- Current character position in the buffer
 * uppercaseFlag   -- Indicates that tokens should be made uppercase.
 * ignoreFlag      -- Indicates whether to ignore space characters.
 */

static string buffer;
static int buflen;
static int cpos;
static bool uppercaseFlag = FALSE;
static bool ignoreFlag = FALSE;

/* Private function prototypes */

static void SkipLeadingSpaces(void);

/* Exported entries */

/*
 * Function: InitScanner
 * -----
 * All this function has to do is initialize the private
 * variables used in the package. The function CopyString
 * is used to ensure that the caller cannot change the
 * contents of the string after calling InitScanner.
 */

void InitScanner(string line)
{
    buffer = CopyString(line);
    buflen = StringLength(buffer);
    cpos = 0;
}
```

```
/*
 * Function: GetNextToken
 * -----
 * The implementation of GetNextToken follows its
 * behavioral description as given in the interface:
 * if the next character is alphanumeric, the next
 * string of such characters is returned; if it is
 * a non-alphanumeric character, only that character
 * is returned. If GetNextToken is called at the
 * end of the line, it returns the empty string,
 * although clients can also test for this condition
 * by calling the predicate function AtEndOfLine.
 */

string GetNextToken(void)
{
    char ch;
    int start;
    string token;


    if (ignoreFlag) SkipLeadingSpaces();
    if (cpos >= buflen) return ("");
    ch = IthChar(buffer, cpos);
    if (isalnum(ch)) {
        start = cpos;
        while (cpos < buflen && isalnum(IthChar(buffer, cpos))) {
            cpos++;
        }
        token = SubString(buffer, start, cpos - 1);
    } else {
        cpos++;
        token = CharToString(ch);
    }
    if (uppercaseFlag) token = ConvertToUpperCase(token);
    return (token);
}

/*
 * Function: AtEndOfLine
 * -----
 * This implementation simply compares the current buffer
 * position against the saved length.
 */

bool AtEndOfLine(void)
{
    if (ignoreFlag) SkipLeadingSpaces();
    return (cpos >= buflen);
}

/*
 * Function: ReturnUppercaseTokens
 * -----
 * This function allows the client to determine whether tokens
 * should be returned in uppercase. The implementation just
 * sets a flag for the other functions.
 */

void ReturnUppercaseTokens(bool flag)
{
    uppercaseFlag = flag;
}
```



```
/*
 * Function: IgnoreSpaces
 * -----
 * This function allows the client to determine whether spaces
 * are valid tokens. The implementation just sets a flag for
 * the other functions.
 */

void IgnoreSpaces(bool flag)
{
    ignoreFlag = flag;
}

/* Private functions */

/*
 * Function: SkipLeadingSpaces
 * Usage: SkipLeadingSpaces();
 * -----
 * This function skips over leading spaces in the buffer
 * and leaves cpos with the index of the next nonspace
 * character.
 */

static void SkipLeadingSpaces(void)
{
    while (cpos < buflen && isspace(IthChar(buffer, cpos))) {
        cpos++;
    }
}
```

9. See the solution to exercises 8.



10.

```
/*
 * File: calc.c
 * -----
 * This program implements a simple calculator that applies
 * arithmetic operators to integers.
 */

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

#include "genlib.h"
#include "strlib.h"
#include "simpio.h"
#include "scanner.h"

/* Private function prototypes */

static void GiveInstructions(void);
static int EvaluateLine(string line);

/* Main program */

main()
{
    string line;

    IgnoreSpaces(TRUE);
    GiveInstructions();
    while (TRUE) {
        printf("> ");
        line = GetLine();
        if (StringEqual(line, "")) break;
        printf("%d\n", EvaluateLine(line));
    }
}

/*
 * Function: GiveInstructions
 * Usage: GiveInstructions();
 * -----
 * This function prints instructions for using the calculator.
 */

static void GiveInstructions(void)
{
    printf("This program implements a simple calculator.\n");
    printf("When the > prompt appears, enter an expression\n");
    printf("consisting of integer constants and the operators\n");
    printf("+, -, *, /, and %. To stop, enter a blank line.\n");
}
```

```
/*
 * Function: EvaluateLine
 * Usage: result = EvaluateLine(line);
 * -----
 * This function takes a line of text consisting of an arithmetic
 * expression and evaluates it, returning the integer result. The
 * expression consists of any number of integer constants separated
 * by one of the following operators: +, -, *, /, %. The operators
 * are evaluated in strictly left-to-right order. Errors in the
 * expression are reported by the Error function, which causes the
 * program to exit.
 */

static int EvaluateLine(string line)
{
    string token, op;
    int result, value;

    InitScanner(line);
    result = StringToInteger(GetNextToken());
    while (!AtEndOfLine()) {
        op = GetNextToken();
        value = StringToInteger(GetNextToken());
        switch (IthChar(op, 0)) {
            case '+': result += value; break;
            case '-': result -= value; break;
            case '*': result *= value; break;
            case '/': result /= value; break;
            case '%': result %= value; break;
            default: Error("Illegal operator %s", op);
        }
    }
    return (result);
}
```

11.

```
/*
 * File: labtest.c
 * -----
 * This program tests the labelseq abstraction by using
 * it to generate a sequence of labels.
 */

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

#include "genlib.h"
#include "strlib.h"
#include "simpio.h"
#include "labelseq.h"

/*
 * Constants
 * -----
 * NLabels -- Number of labels to generate.
 */

#define NLabels 5

/* Main program */

main()
{
    string prefix;
    int i, start;

    printf("This program tests the labelseq abstraction.\n");
    printf("Prefix to use for labels: ");
    prefix = GetLine();
    printf("Starting number: ");
    start = GetInteger();
    SetLabelPrefix(prefix);
    SetLabelNumber(start);
    for (i = 0; i < NLabels; i++) {
        printf("%s\n", GetNextLabel());
    }
}
```

```
/*
 * File: labelseq.h
 * -----
 * This interface provides a mechanism for generating
 * a series of distinct identifiers that the client can
 * then use as tags for some internal value.
 */

#ifndef _labelseq_h
#define _labelseq_h

#include "genlib.h"

/*
 * Function: GetNextLabel
 * Usage: label = GetNextLabel();
 * -----
 * This function returns the next identifier in the sequence.
 */

string GetNextLabel(void);

/*
 * Function: SetLabelPrefix
 * Usage: SetLabelPrefix(prefix);
 * -----
 * This function sets the prefix for the labelseq package
 * so that subsequent labels begin with this string. The
 * default prefix for labels is the string "Label".
 */

void SetLabelPrefix(string prefix);

/*
 * Function: SetLabelNumber
 * Usage: SetLabelNumber(number);
 * -----
 * This function sets the sequence number for the next label.
 * By default, this package generates labels starting with
 * sequence number 1.
 */

void SetLabelNumber(int number);

#endif
```

```
/*
 * File: labelseq.c
 * -----
 * This file implements the labelseq.h interface.
 */

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

#include "genlib.h"
#include "strlib.h"
#include "labelseq.h"

/*
 * Private variables
 * -----
 * labelPrefix -- Prefix string used for each label
 * labelNumber -- Sequence number of the next label
 */

static string labelPrefix = "Label";
static int labelNumber = 1;

/* Exported entries */

string GetNextLabel(void)
{
    string label;

    label = Concat(labelPrefix, IntegerToString(labelNumber));
    labelNumber++;
    return (label);
}

void SetLabelPrefix(string prefix)
{
    labelPrefix = prefix;
}

void SetLabelNumber(int number)
{
    labelNumber = number;
}
```

12.

```
/*
 * File: primes.c
 * -----
 * This file lists the primes between two limits. The program
 * uses the fill.h interface to print the results as a
 * continuous stream, breaking lines only when necessary.
 */

#include <stdio.h>
#include <math.h>
#include "genlib.h"
#include "strlib.h"
#include "fill.h"

/*
 * Constants
 * -----
 * LowerLimit -- Starting value for the prime search
 * UpperLimit -- Final value for the prime search
 * FillMargin -- Column at which each line should end
 */

#define LowerLimit    0
#define UpperLimit    200
#define FillMargin    55

/* Function prototypes */

bool IsPrime(int n);

/* Main program */

main()
{
    bool commaFlag;
    int i;

    SetFillMargin(FillMargin);
    PrintFilledString("The primes between ");
    PrintFilledString(IntegerToString(LowerLimit));
    PrintFilledString(" and ");
    PrintFilledString(IntegerToString(UpperLimit));
    PrintFilledString(" are: ");
    commaFlag = FALSE;
    for (i = LowerLimit; i <= UpperLimit; i++) {
        if (IsPrime(i)) {
            if (commaFlag) PrintFilledString(", ");
            PrintFilledString(IntegerToString(i));
            commaFlag = TRUE;
        }
    }
    PrintFilledString(".\n");
}

/* The IsPrime function appears in Chapter 6 of the text. */
```

```
/*
 * fill.c
 * -----
 * This file implements the fill.h interface.
 */

#include <stdio.h>
#include "genlib.h"
#include "strlib.h"
#include "fill.h"

/*
 * Private variables
 * -----
 * buffer      -- The string being saved for output
 * spaceCount  -- Number of spaces to print before string
 * linePosition -- Current position on the line
 * fillMargin  -- Right margin
 */


static string buffer = "";
static int fillMargin = 65;
static int spaceCount = 0;
static int linePosition = 0;

/* Private function prototypes */
static void PrintBuffer(void);

/* Exported entries */
void SetFillMargin(int margin)
{
    fillMargin = margin;
}

void PrintFilledString(string str)
{
    int i;
    char ch;

    for (i = 0; i < StringLength(str); i++) {
        ch = IthChar(str, i);
        switch (ch) {
            case ' ':
                if (StringLength(buffer) == 0) {
                    spaceCount++;
                } else {
                    PrintBuffer();
                    spaceCount = 1;
                }
                break;
            case '\n':
                PrintBuffer();
                printf("\n");
                spaceCount = 0;
                linePosition = 0;
                break;
            default:
                buffer = Concat(buffer, CharToString(ch));
                break;
        }
    }
}
```



```
static void PrintBuffer(void)
{
    int i, len;

    len = StringLength(buffer);
    if (linePosition > 0
        && len + spaceCount + linePosition > fillMargin) {
        printf("\n");
        spaceCount = 0;
        linePosition = 0;
    }
    for (i = 0; i < spaceCount; i++) {
        printf(" ");
    }
    printf("%s", buffer);
    linePosition += len + spaceCount;
    buffer = "";
    spaceCount = 0;
}
```



## Solutions for Chapter 11

### Arrays

#### Review questions

1. An array is ordered and homogeneous.
2. An *element* of an array is one of its ordered components. The position number of an element in an array is called its *index*; in C, all index numbers for arrays begin at 0. The data type of each element within an array is the *element type* for the array; because arrays are homogeneous, the element types within a single array must always be the same. The *array size* is the number of elements allocated to an array in its declaration. *Selection* is the process of identifying a particular array element by its index number.
3.
  - a. 

```
#define MaxRealArray 100
double realArray[MaxRealArray];
```
  - b. 

```
#define MaxInUse 16
bool inUse[MaxInUse];
```
  - c. 

```
#define MaxLines 1000
string lines[MaxLines];
```
4. 

```
#define NSquares 11
int squares[NSquares];

int i;
for (i = 0; i < NSquares; i++) {
    squares[i] = i * i;
}
```
5. The first approach is to use zero-based indices internally and then add 1 to the internal index whenever the index value is displayed to the user. The second is to allocate an extra element and then ignore element 0 of the array. The first strategy allows you to use functions that work with zero-based arrays directly; the second strategy usually is easier to understand because the program and the user both work with the same set of indices.
6. A *bit* is the smallest possible unit of information and has exactly two states, usually represented as 0 and 1. A *byte* is a collection of bits, usually eight, large enough to store a character value; a *word* is a larger collection of information capable of holding an integer value. An *address* is the numeric memory location at which a particular value is stored.
7. The `sizeof` operator returns the number of bytes required to represent a data value in C.
8. 20.
9. `NElements * sizeof(double)`
10. On most machines, selecting a value outside the allocated bounds returns the contents of the memory where that element would be placed, even though the array does not in fact contain that element. Because you have no easy way to know what data the compiler has assigned to that memory location, the results are completely unpredictable.
11. The allocated size of an array is the total number of elements in the array at the time of its declaration. In many cases, however, programs use only part of the allocated space. The effective size is the number of elements that are actually in use.
12. If you call a function that takes an array as a formal parameter, the array storage used for the parameter is shared with that of the actual argument. The major implication of this rule is that changing the value of an element of the parameter array changes the value of the corresponding element in the argument array.

13. When an array variable is used as an argument, the formal parameter is assigned the address of the initial element in that array. Because the address is in fact that of the calling array, selecting an element from a formal parameter array identifies a memory location within the argument array.
14. True. If the array is a multidimensional array, only the first subscript bound can be omitted.
15. The variable `tmp` in the `SwapIntegerElements` function holds the original value of one of the array elements being exchanged. If you store a value in an array element without first making sure that you have saved the old value, there is no way to complete the exchange operation.
16. `static int squares[] = { 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 };`
17. `sizeof array / sizeof array[0]`
18. `static string directionNames[] = { "North", "East", "South", "West" };  
printf("%s\n", directionNames[dir]);`
19. In C, a multidimensional array is an array whose elements are other arrays. Conceptually, the elements of a multidimensional array are identified by several index numbers specifying, for example, the row and column of an element.
- 20.

|      |       |                                |
|------|-------|--------------------------------|
| 1000 | ----- | <code>rectangular[0][0]</code> |
| 1002 | ----- | <code>rectangular[0][1]</code> |
| 1004 | ----- | <code>rectangular[0][2]</code> |
| 1006 | ----- | <code>rectangular[1][0]</code> |
| 1008 | ----- | <code>rectangular[1][1]</code> |
| 1010 | ----- | <code>rectangular[1][2]</code> |

21. `static char chessboard[8][8] = {  
' r', ' n', ' b', ' q', ' k', ' b', ' n', ' r',  
' p', ' p', ' p', ' p', ' p', ' p', ' p', ' p',  
' -', ' -', ' -', ' -', ' -', ' -', ' -', ' -',  
' -', ' -', ' -', ' -', ' -', ' -', ' -', ' -',  
' -', ' -', ' -', ' -', ' -', ' -', ' -', ' -',  
' -', ' -', ' -', ' -', ' -', ' -', ' -', ' -',  
' P', ' P', ' P', ' P', ' P', ' P', ' P', ' P',  
' R', ' N', ' B', ' Q', ' K', ' B', ' N', ' R',  
};`

## Programming exercises

1.

```
/*
 * File: sizeof.c
 * -----
 * This program displays the sizes of each of the built-in types.
 * This program will have different results on different computer
 * systems because the size of individual types differs from machine
 * to machine.
 */

#include <stdio.h>
#include "genlib.h"

/* Private function prototypes */

static void DisplaySize(string name, int size);

/* Main program */

main()
{
    DisplaySize("char",    sizeof(char));
    DisplaySize("int",     sizeof(int));
    DisplaySize("short",   sizeof(short));
    DisplaySize("long",    sizeof(long));
    DisplaySize("float",   sizeof(float));
    DisplaySize("double",  sizeof(double));
}

/*
 * Function: DisplaySize
 * Usage: DisplaySize(name, size);
 * -----
 * This function prints out each line of the size listing.
 */

static void DisplaySize(string name, int size)
{
    printf("Values of type %s require %d", name, size);
    printf(" byte%s.\n", (size == 1) ? "" : "s");
}
```

2.

```
/*
 * File: gymscore.c
 * -----
 * This program computes a gymnastic score by dropping the
 * lowest and highest scores and then averaging the remainder.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constants
 * -----
 * NJudges -- Number of judges
 */

#define NJudges 7

/* Private function prototypes */

static void ReadScores(double scores[], int nScores);
static double CompositeScore(double scores[], int nScores);

/* Main program */


main()
{
    double scores[NJudges], composite;

    printf("This program computes a composite gymnastics score.\n");
    ReadScores(scores, NJudges);
    composite = CompositeScore(scores, NJudges);
    printf("The composite score is %.1f\n", composite);
}

/*
 * Function: ReadScores
 * Usage: ReadScores(scores, nScores);
 * -----
 * This function allows the user to enter the scores for a set
 * of judges into the array scores. The number of scores is
 * given by the parameter nScores.
 */

static void ReadScores(double scores[], int nScores)
{
    int i;

    printf("Enter the scores for each judge.\n");
    for (i = 0; i < nScores; i++) {
        printf("Judge #%d: ", i + 1);
        scores[i] = GetReal();
    }
}
```



```
/*
 * Function: CompositeScore
 * Usage: score = CompositeScore(scores, nScores);
 * -----
 * This function returns the composite score given a set of
 * judging scores. The composite is the average score after
 * throwing out the highest and lowest score. The parameter
 * nScores gives the number of scores and must be at least 3.
 */

static double CompositeScore(double scores[], int nScores)
{
    double largest, smallest, total;
    int i;

    if (nScores < 3) Error("Too few scores");
    largest = smallest = total = scores[0];
    for (i = 1; i < nScores; i++) {
        if (scores[i] > largest) largest = scores[i];
        if (scores[i] < smallest) smallest = scores[i];
        total += scores[i];
    }
    return ((total - largest - smallest) / (nScores - 2));
}
```

3.

```
/*
 * File: mean.c
 * -----
 * This program averages a set of five gymnastic scores. It
 * is similar to the program given in Chapter 11 except in
 * two respects: (1) the judges are numbered from 1 to 5
 * instead of from 0 to 4 and (2) a separate Mean function
 * is used to calculate the mean.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constants
 * -----
 * NJudges -- Number of judges
 */

#define NJudges 5

/* Private function prototypes */

static double Mean(double array[], int n);

/* Main program */

main()
{
    double scores[NJudges];
    int i;

    printf("Please enter a score for each judge.\n");
    for (i = 0; i < NJudges; i++) {
        printf("Score for judge #%d: ", i + 1);
        scores[i] = GetReal();
    }
    printf("The average score is %.2f\n", Mean(scores, NJudges));
}

/*
 * Function: Mean
 * Usage: mean = Mean(array, n);
 * -----
 * This function returns the statistical mean (average) of a
 * distribution stored in array, which has effective size n.
 */

static double Mean(double array[], int n)
{
    int i;
    double total;

    total = 0;
    for (i = 0; i < n; i++) {
        total += array[i];
    }
    return (total / n);
}
```

4.

```

/*
 * File: stddev.c
 * -----
 * This program computes the standard deviation of a distribution.
 */

#include <stdio.h>
#include <math.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constants
 * -----
 * MaxElements -- The maximum number of elements in the array
 * Sentinel    -- Sentinel value used to indicate end of input
 */

#define MaxElements 100
#define Sentinel    -1

/* Private function prototypes */

static double StandardDeviation(double array[], int n);
static double Mean(double array[], int n);
static int GetRealArray(double array[], int max, double sentinel);

/* Main program */

main()
{
    double array[MaxElements], stddev;
    int n;

    printf("This program finds the standard deviation of an array.\n");
    printf("Enter the elements, one per line.\n");
    printf("Use %g to signal the end of list.\n", (double) Sentinel);
    n = GetRealArray(array, MaxElements, Sentinel);
    stddev = StandardDeviation(array, n);
    printf("The standard deviation is %g\n", stddev);
}

/*
 * Function: StandardDeviation
 * Usage: stddev = StandardDeviation(array, n);
 * -----
 * This function returns the standard deviation of the array.
 * The parameter n specifies the effective size.
 */

static double StandardDeviation(double array[], int n)
{
    double mean, total, delta;
    int i;

    mean = Mean(array, n);
    total = 0;
    for (i = 0; i < n; i++) {
        delta = array[i] - mean;
        total += delta * delta;
    }
    return (sqrt(total / n));
}

```

```
/*
 * Function: Mean
 * Usage: mean = Mean(array, n);
 * -----
 * This function returns the statistical mean (average) of a
 * distribution stored in array, which has effective size n.
 */

static double Mean(double array[], int n)
{
    int i;
    double total;

    total = 0;
    for (i = 0; i < n; i++) {
        total += array[i];
    }
    return (total / n);
}

/*
 * Function: GetRealArray
 * Usage: n = GetRealArray(array, max, sentinel);
 * -----
 * This function reads elements into an array of doubles by
 * reading values, one per line, from the keyboard. The end
 * of the input data is indicated by the parameter sentinel.
 * The caller is responsible for declaring the array and
 * passing it in as a parameter, along with its allocated
 * size. The value returned is the number of elements
 * actually entered and therefore gives the effective size
 * of the array, which is typically less than the allocated
 * size given by max. If the user types in more than max
 * elements, GetRealArray generates an error.
 */

static int GetRealArray(double array[], int max, double sentinel)
{
    int n;
    double value;

    n = 0;
    while (TRUE) {
        printf(" ? ");
        value = GetReal();
        if (value == sentinel) break;
        if (n == max) Error("Too many input items for array");
        array[n] = value;
        n++;
    }
    return (n);
}
```



5.

```
/*
 * File: sieve.c
 * -----
 * This program computes primes less than or equal to UpperLimit
 * using an algorithm called the "Sieve of Eratosthenes".
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constants
 * -----
 * UpperLimit -- Maximum value to include in the list
 */

#define UpperLimit 1000

/* Main program */

main()
{
    bool candidate[UpperLimit + 1];
    int n, k;

    printf("This program lists the primes less than or equal to %d\n",
           UpperLimit);
    for (n = 2; n <= UpperLimit; n++) {
        candidate[n] = TRUE;
    }
    for (n = 2; n <= UpperLimit; n++) {
        if (candidate[n]) {
            printf("%d\n", n);
            for (k = 2 * n; k <= UpperLimit; k += n) {
                candidate[k] = FALSE;
            }
        }
    }
}
```

6.

```


/*
 * File: morse.c
 * -----
 * This program translates a line of text into Morse code.
 * Only the alphabetic characters are considered. Translation
 * is performed by using the character as an index into a
 * translation array.
 */

#include <stdio.h>
#include <ctype.h>
#include "genlib.h"
#include "simpio.h"
#include "strlib.h"

/*
 * Global variable: MorseCodeTable
 * -----
 * The elements of this table contain the Morse code values
 * for the 26 letters. The table is indexed by the ordinal
 * value of a letter (A = 0, B = 1, etc.). This index is
 * computed by subtracting the ASCII code for 'A' from the
 * uppercase character.
 */

static const string MorseCodeTable[26] = {
    "-.-"      /* A */,
    "-..."   /* B */,
    "-.-.-"    /* C */,
    "-.-"      /* D */,
    "."        /* E */,
    "..-.-"    /* F */,
    "---"      /* G */,
    "...."     /* H */,
    ".."       /* I */,
    ".---"     /* J */,
    "-.-"      /* K */,
    "-.-.-"    /* L */,
    "---"      /* M */,
    "-."       /* N */,
    "---"      /* O */,
    ".-.-"     /* P */,
    "-.-.-"    /* Q */,
    "-.-"      /* R */,
    "...."     /* S */,
    "-"        /* T */,
    "..-.-"    /* U */,
    "....-"    /* V */,
    "-.-"      /* W */,
    "-.-.-"    /* X */,
    "-.-.-"    /* Y */,
    "-.-.-"    /* Z */
};

```



```
/* Main program */

main()
{
    string line;
    int i;
    char ch;

    printf("This program translates a line into Morse code.\n");
    printf("Enter English text: ");
    line = GetLine();
    for (i = 0; i < StringLength(line); i++) {
        ch = IthChar(line, i);
        if (ch == ' ') printf("\n");
        if (isalpha(ch)) {
            printf("%s ", MorseCodeTable[toupper(ch) - 'A']);
        }
    }
    printf("\n");
}
```

7. This problem is solved as part of the more general program in exercise 8.

8.

```

/*
 * File: testhist.c
 * -----
 * This program tests the histogram function. The test
 * program is responsible for reading in the data and then
 * calling the program to print the histogram.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"
#include "hist.h"

/*
 * Constants
 * -----
 * MaxScores -- Maximum number of scores
 * Sentinel   -- Value used to terminate input
 * LowScore   -- Lowest score in histogram
 * HighScore  -- Highest scores in histogram
 * RangeSize  -- Number of scores per range
 */

#define MaxScores 200
#define Sentinel -1
#define LowScore 0
#define HighScore 100
#define RangeSize 10

/* Private function prototypes */

static void PrintInstructions(void);
static int GetIntegerArray(int array[], int max, int sentinel);

/* Main program */

main()
{
    int scores[MaxScores];
    int nScores;

    PrintInstructions();
    nScores = GetIntegerArray(scores, MaxScores, Sentinel);
    GenerateHistogram(scores, nScores, LowScore, HighScore, RangeSize);
}

/*
 * Function: PrintInstructions
 * Usage: PrintInstruction();
 * -----
 * This function displays instructions for using the test program.
 */

static void PrintInstructions(void)
{
    printf("This program prints a histogram of exam scores.\n");
    printf("Enter the list of exam scores, one per line. Signal\n");
    printf("the end of the list by entering %d.\n", Sentinel);
}

/* The GetIntegerArray function is given in the text */

```

```
/*
 * File: hist.h
 * -----
 * The hist.h file is the interface to a histogram package
 * that allows clients to generate a histogram of elements
 * in an integer array.
 */

#ifndef _hist_h
#define _hist_h

/*
 * Function: GenerateHistogram
 * Usage: GenerateHistogram(array, n, low, high, rangeSize);
 * -----
 * This function writes out a histogram of the values stored
 * in the array. Each line in the histogram consists of a
 * row of stars for each element in the array that falls
 * within one of the selected ranges. Each range consists
 * of rangeSize numbers, starting at low, and going up to
 * high (the last range may be shorter than the others).
 * The parameter n gives the effective size of the array.
 * Any data values that do not fall within the range between
 * low and high are simply ignored.
 */

void GenerateHistogram(int array[], int n,
                      int low, int high, int rangeSize);

#endif
```

```
/*
 * File: hist.c
 * -----
 * This file implements the hist.h interface.
 */

#include <stdio.h>
#include "genlib.h"
#include "strlib.h"
#include "hist.h"

/*
 * Constants
 * -----
 * MaxRanges -- Maximum number of ranges in the histogram
 * HistChar   -- Character used to indicate each value
 */

#define MaxRanges 101
#define HistChar  '*'

/* Private function declarations */

static int NumberOfRanges(int low, int high, int rangeSize);
static int ComputeRange(int value, int low, int high, int rangeSize);
static void PrintHistogram(int ranges[], int nRanges,
                           int low, int high, int rangeSize);
static void PrintHistogramLabel(int bottom, int top);
static void PrintNCharacters(int n, char ch);
static void ClearIntegerArray(int array[], int n);

/* Public entry */

/*
 * Function: GenerateHistogram
 * Implementation notes (see interface for usage)
 * -----
 * The function begins by initializing a new array whose elements each
 * correspond to a range of values in the data array. The function then
 * goes through each element in the data array and figures out which
 * range that element falls into. For each data value, the function then
 * increments the count stored in the appropriate element of rangeArray.
 * After going through all the elements, the function then goes through
 * the ranges and displays the histogram.
 */

void GenerateHistogram(int array[], int n,
                      int low, int high, int rangeSize)
{
    int rangeArray[MaxRanges];
    int i, range, nRanges;

    nRanges = NumberOfRanges(low, high, rangeSize);
    if (nRanges > MaxRanges) Error("Too many ranges");
    ClearIntegerArray(rangeArray, nRanges);
    for (i = 0; i < n; i++) {
        range = ComputeRange(array[i], low, high, rangeSize);
        if (range != -1) rangeArray[range]++;
    }
    PrintHistogram(rangeArray, nRanges, low, high, rangeSize);
}
```

```

/* Private functions */

/*
 * Function: NumberOfRanges
 * Usage: nRanges = NumberOfRanges(low, high, rangeSize);
 * -----
 * This function computes the number of ranges in the histogram
 * given the input parameters.
 */

static int NumberOfRanges(int low, int high, int rangeSize)
{
    return (((high - low) / rangeSize) + 1);
}

/*
 * Function: ComputeRange
 * Usage: b = ComputeRange(value, low, high, rangeSize);
 * -----
 * This function determines which range number the value should
 * go into in the histogram. The value is determined by linear
 * scaling, just as in the random number package. If the
 * value is outside the range, ComputeRange returns -1.
 * It is the client's responsibility to check for this
 * result.
 */

static int ComputeRange(int value, int low, int high, int rangeSize)
{
    if (value < low || value > high) return (-1);
    return ((value - low) / rangeSize);
}

/*
 * Function: PrintHistogram
 * Usage: PrintHistogram(rangeArray, nRanges, low, high, rangeSize);
 * -----
 * This function displays the histogram using the data stored in the
 * array rangeArray.
 */

static void PrintHistogram(int rangeArray[], int nRanges,
                           int low, int high, int rangeSize)
{
    int range, bottom, top;

    bottom = low;
    for (range = 0; range < nRanges; range++) {
        top = bottom + rangeSize - 1;
        if (top > high) top = high;
        PrintHistogramLabel(bottom, top);
        PrintNCharacters(rangeArray[range], HistChar);
        printf("\n");
        bottom += rangeSize;
    }
}

```

```
/*
 * Function: PrintHistogramLabel
 * Usage: PrintHistogramLabel(bottom, top);
 * -----
 * This function displays the label on the histogram line.
 */

static void PrintHistogramLabel(int bottom, int top)
{
    string label;

    if (bottom == top) {
        label = IntegerToString(bottom);
    } else {
        label = Concat(IntegerToString(bottom),
                       Concat("-", IntegerToString(top)));
    }
    printf("%-5s | ", label);
}

/*
 * Function: PrintNCharacters
 * Usage: PrintNCharacters(n, ch);
 * -----
 * This function prints n copies of the character ch.
 */

static void PrintNCharacters(int n, char ch)
{
    int i;

    for (i = 0; i < n; i++) printf("%c", ch);
}

/*
 * Function: ClearIntegerArray
 * Usage: ClearIntegerArray(array, n);
 * -----
 * This function sets the first n elements in the array to 0.
 */

static void ClearIntegerArray(int array[], int n)
{
    int i;

    for (i = 0; i < n; i++) {
        array[i] = 0;
    }
}
```



9.

```
/*
 * File: lineplot.c
 * -----
 * This program plots a line connecting a set of x, y
 * coordinate values.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"
#include "strlib.h"
#include "graphics.h"

/*
 * Constants
 * -----
 * MaxPoints    -- Maximum number of points to be plotted
 * PointRadius   -- Radius of the circle indicating a data point
 */

#define MaxPoints    100
#define PointRadius  (1 / 72.0)

/* Private function prototypes */

static int GetPoints(double xCoords[], double yCoords[], int max);
static void DrawLineGraph(double xCoords[], double yCoords[], int n);
static void DrawCenteredCircle(double x, double y, double r);
static void DrawLineTo(double x, double y);

/* Main program */

main()
{
    double xCoords[MaxPoints], yCoords[MaxPoints];
    int n;

    InitGraphics();
    printf("This program plots a set of points.\n");
    printf("Enter the points as x, y, ending with a blank line.\n");
    n = GetPoints(xCoords, yCoords, MaxPoints);
    DrawLineGraph(xCoords, yCoords, n);
}
```

```

/*
 * Function: GetPoints
 * Usage: n = GetPoints(xCoords, yCoords, max);
 * -----
 * This function reads in a set of points from the user, represented
 * as pairs of x and y values separated by commas. A blank line is
 * used as a sentinel to indicate the end of input. The x-coordinates
 * of the points are stored in the array xCoords; the y-coordinates
 * are stored in the corresponding entry of the array yCoords. The
 * caller is responsible for declaring these arrays and for assigning
 * enough space to hold the input data. The parameter max specifies
 * the allocated size of the array; if the user attempts to enter more
 * points than allowed by this maximum, an error occurs. The function
 * returns the number of points entered.
 */

static int GetPoints(double xCoords[], double yCoords[], int max)
{
    string line;
    int n, comma;

    n = 0;
    while (TRUE) {
        printf("> ");
        line = GetLine();
        if (StringEqual(line, "")) break;
        comma = FindChar(',', line, 0);
        if (comma == -1) Error("Illegal point specified");
        if (n >= max) Error("Too many points");
        xCoords[n] = StringToReal(SubString(line, 0, comma - 1));
        yCoords[n] = StringToReal(
            SubString(line, comma + 1, StringLength(line) - 1));
        n++;
    }
    return (n);
}

/*
 * Function: DrawLineGraph
 * Usage: DrawLineGraph(xCoords, yCoords, n);
 * -----
 * This function draws a line graph connecting the points whose
 * coordinates are specified in the first n elements of the arrays
 * xCoords and yCoords. Each point is displayed as a small circle,
 * and each adjacent pair of points is connected by a straight line.
 */

static void DrawLineGraph(double xCoords[], double yCoords[], int n)
{
    int i;

    for (i = 0; i < n; i++) {
        if (i > 0) DrawLineTo(xCoords[i], yCoords[i]);
        DrawCenteredCircle(xCoords[i], yCoords[i], PointRadius);
        MovePen(xCoords[i], yCoords[i]);
    }
}

/* DrawCenteredCircle and DrawLineTo are given in the text. */

```

10.

```

/*
 * File: probtest.c
 * -----
 * This program executes the probability experiment of dropping
 * a marble through a triangular grid of pegs. At each peg, the
 * marble falls either left or right.
 */

#include <stdio.h>
#include "genlib.h"
#include "random.h"
#include "graphics.h"

/*
 * Constants
 * -----
 * NTrials          -- Number of marbles to drop
 * NColumns         -- Number of columns at the bottom
 * MarbleRadius     -- Radius of the marble
 * MarbleSeparation -- Vertical separation between marbles
 * ColumnWidth      -- Width of the column
 * ColumnHeight     -- Height of the column
 */

#define NTrials          50
#define NColumns         10
#define MarbleRadius     (4.0 / 72.0)
#define MarbleSeparation (1.0 / 72.0)
#define ColumnWidth      (12.0 / 72.0)
#define ColumnHeight     2.25

/* Private function prototypes */

static int ChooseColumn(int nColumns);
static void DisplayMarble(int column, int height);
static void DrawColumns(void);
static void ClearIntegerArray(int array[], int n);
static void DrawCenteredCircle(double x, double y, double r);

/* Main program */

main()
{
    int columnHeight[NColumns];
    int i, column;

    InitGraphics();
    Randomize();
    DrawColumns();
    ClearIntegerArray(columnHeight, NColumns);
    for (i = 0; i < NTrials; i++) {
        column = ChooseColumn(NColumns);
        DisplayMarble(column, columnHeight[column]);
        columnHeight[column]++;
    }
}

```

```
/*
 * Function: ChooseColumn
 * Usage: column = ChooseColumn(nColumns);
 * -----
 * This function simulates a ball dropping through a probability
 * board in which the ball can bounce left or right at each step.
 * This function simplifies the computation by taking advantage of
 * the fact that the operation of the traditional grid of bounce
 * points is the same as one in which the pegs are arranged like
 * this:
 *
 *      o
 *     oo
 *    ooo
 *   oooo
 *
 * and the ball either falls straight down or moves one unit to
 * to the right.
 */

static int ChooseColumn(int nColumns)
{
    int x, y;

    x = 0;
    for (y = 0; y < nColumns - 1; y++) {
        if (RandomChance(0.5)) x++;
    }
    return (x);
}

/*
 * Function: DisplayMarble
 * Usage: DisplayMarble(column, height);
 * -----
 * This function draws a circle to represent a marble in the
 * specified column. The parameter height gives the number of
 * marbles in that column before this marble is added.
 */

static void DisplayMarble(int column, int height)
{
    double x, y;

    x = (column + 0.5) * ColumnWidth;
    y = (2 * MarbleRadius + MarbleSeparation) * (height + 0.5);
    DrawCenteredCircle(x, y, MarbleRadius);
}
```

```
/*
 * Function: DrawColumns
 * Usage: DrawColumns();
 * -----
 * This function draws the dividing lines forming the columns.
 */

static void DrawColumns(void)
{
    int i;

    for (i = 1; i <= NColumns; i++) {
        MovePen(i * ColumnWidth, 0);
        DrawLine(0, ColumnHeight);
    }
}

/*
 * Function: ClearIntegerArray
 * Usage: ClearIntegerArray(array, n);
 * -----
 * This function sets the first n elements in the array to 0.
 */

static void ClearIntegerArray(int array[], int n)
{
    int i;

    for (i = 0; i < n; i++) {
        array[i] = 0;
    }
}

/* The function DrawCenteredCircle appears in Chapter 7 */
```

11.

```
/*
 * Function: IsWinningPosition
 * Usage: if (IsWinningPosition(board, player)) . . .
 * -----
 * This function returns TRUE if the specified player has won the game
 * represented by board. The program operates by checking all of the
 * winning paths. The work of this function is subdivided among three
 * subsidiary functions that check the rows, columns, and diagonals.
 */

static bool IsWinningPosition(char board[3][3], char player)
{
    return (CheckRows(board, player)
           || CheckColumns(board, player)
           || CheckDiagonals(board, player));
}


/*
 * Functions: CheckRows, CheckColumns, CheckDiagonals
 * Usage: flag = CheckRows(board, player);
 *        flag = CheckColumns(board, player);
 *        flag = CheckDiagonals(board, player);
 * -----
 * This function returns TRUE if the specified player has three symbols
 * in the specified alignment.
 */

static bool CheckRows(char board[3][3], char player)
{
    int row, column, count;

    for (row = 0; row < 3; row++) {
        count = 0;
        for (column = 0; column < 3; column++) {
            if (board[row][column] == player) count++;
        }
        if (count == 3) return (TRUE);
    }
    return (FALSE);
}

static bool CheckColumns(char board[3][3], char player)
{
    int row, column, count;

    for (column = 0; column < 3; column++) {
        count = 0;
        for (row = 0; row < 3; row++) {
            if (board[row][column] == player) count++;
        }
        if (count == 3) return (TRUE);
    }
    return (FALSE);
}
```



```
static bool CheckDiagnals(char board[3][3], char player)
{
    int row, count;

    count = 0;
    for (row = 0; row < 3; row++) {
        if (board[row][row] == player) count++;
    }
    if (count == 3) return (TRUE);
    count = 0;
    for (row = 0; row < 3; row++) {
        if (board[2 - row][row] == player) count++;
    }
    return (count == 3);
}
```

12.

```
/*
 * File: checkers.c
 * -----
 * This program initializes and displays a checkerboard. The
 * checkerboard itself is a two-dimensional array of characters,
 * indexed by row and column. Red checkers are indicated by the
 * letter 'r'; black checkers are indicated by the letter 'b'.
 * Empty squares are indicated using either a space or a '-'
 * depending on the color of the board square.
 */

#include <stdio.h>
#include "genlib.h"

/* Private function prototypes */

static void DisplayCheckerboard(char board[8][8]);
static void InitCheckerboard(char board[8][8]);
static char InitialContents(int row, int column);
static bool IsOdd(int n);

/* Main program */

main()
{
    char checkerboard[8][8];

    InitCheckerboard(checkerboard);
    DisplayCheckerboard(checkerboard);
}

/*
 * Function: DisplayCheckerboard
 * Usage: DisplayCheckerboard(board);
 * -----
 * This function displays the checkerboard. Because the rows
 * are traditionally numbered from the bottom up, this function
 * prints the rows in reverse order.
 */

static void DisplayCheckerboard(char board[8][8])
{
    int row, column;
    char ch;

    for (row = 7; row >= 0; row--) {
        for (column = 0; column < 8; column++) {
            printf(" %c", board[row][column]);
        }
        printf("\n");
    }
}
```



```
/*
 * Function: InitCheckerboard
 * Usage: InitCheckerboard(board);
 * -----
 * This function initializes board so that it holds the starting
 * configuration of a checkers game. The real work is performed
 * by the InitialContents function, which determines the contents
 * of a given square.
 */

static void InitCheckerboard(char board[8][8])
{
    int row, column;

    for (row = 0; row < 8; row++) {
        for (column = 0; column < 8; column++) {
            board[row][column] = InitialContents(row, column);
        }
    }
}

/*
 * Function: InitialContents
 * Usage: ch = InitialContents(row, column);
 * -----
 * This function returns the initial contents given the coordinates
 * of the checkerboard square. White squares (which are easily
 * identified because the sum of their row and column number is
 * odd) are always empty, as indicated by a blank space. The
 * contents of a black square depend on the row. The first three
 * rows contain red checkers, the last three contain black checkers,
 * and the two middle rows are empty.
 */

static char InitialContents(int row, int column)
{
    if (IsOdd(row + column)) return (' ');
    switch (row) {
        case 0: case 1: case 2: return ('r');
        case 3: case 4: return ('-');
        case 5: case 6: case 7: return ('b');
    }
}

/*
 * Function: IsOdd
 * Usage: if (IsOdd(n)) . . .
 * -----
 * This function returns TRUE if n is odd.
 */

static bool IsOdd(int n)
{
    return (n % 2 != 0);
}
```

## Solutions for Chapter 12

### Searching and Sorting

#### Review questions

1. Searching is the process of finding a particular value in an array. Sorting is the process of rearranging the elements of an array so that they appear in some well-defined order.
2. The only changes are in the prototype, where the function name and the types of the first two parameters need to be changed as follows:

```
static int FindRealInArray(double key, double array[], int n);
```

3. Two arrays are parallel if the indices in those arrays define a relationship between the items.
4. The linear search algorithm goes through every element of an array in order to check for the key. The binary search algorithm applies only to sorted arrays and begins by checking the element that is at the halfway point of the array. If the key is smaller than that element, the key can only occur in the first half of the array. Similarly, if the key is larger than the middle element, the key can only be in the second half. In either case, you can disregard half of the array and repeat the process on the remaining subarray.
5. True.
6. Binary search requires that the array be sorted.
7. Conceptually, the selection sort algorithm consists of two nested loops. The outer loop goes through each element position of the array in order from first to last. The inner loop then finds the smallest element between that position and the end of the array. Once the smallest element has been identified, the algorithm proceeds by exchanging the smallest element with the index position marked by the outer loop. After cycle  $i$  of the outer loop, the elements in the array up to position  $i$  are in the correct positions.
8. The function would still work correctly. Once the first  $n - 1$  elements have been correctly positioned, the remaining element—which is now guaranteed to be the largest—must also be in the correct position.
9. `(double) clock() / CLOCKS_PER_SEC`
10. An algorithm is quadratic if the time required to execute it grows in proportion to the square of the size of the input.

## Programming exercises

1.

```

/*
 * File: resistor.c
 * -----
 * This program interprets the resistor color code.
 * Each colored band on a resistor corresponds to a number.
 * The first two bands indicate the first two digits of the
 * resistance; the third band is used to indicate a power of
 * ten by which the first two digits are multiplied. For
 * example, the sequence yellow-violet-orange (4-7-3)
 * corresponds to:
 *
 *           3
 *         47 x 10
 */

#include <stdio.h>
#include <math.h>
#include "genlib.h"
#include "simpio.h"
#include "strlib.h"

/*
 * Global variable: colorTable
 * -----
 * This table stores the names of the colors in a string
 * array where the index of the entry corresponds to the
 * digit 0-9.
 */

static string colorTable[] = {
    "BLACK", "BROWN", "RED", "ORANGE", "YELLOW",
    "GREEN", "BLUE", "VIOLET", "GRAY", "WHITE"
};

static int nColors = sizeof colorTable / sizeof colorTable[0];

/* Private function prototypes */

static int GetColorBand(string prompt);
static int FindStringInArray(string key, string array[], int n);

/* Main program */

main()
{
    int band1, band2, band3;
    double resistance;

    printf("This program interprets the resistor color code.\n");
    band1 = GetColorBand("Color of first band: ");
    band2 = GetColorBand("Color of second band: ");
    band3 = GetColorBand("Color of third band: ");
    resistance = (10 * band1 + band2) * pow(10.0, band3);
    printf("Resistance = %g ohms.\n", resistance);
}

```

```
/*
 * Function: GetColorBand
 * Usage: n = GetColorBand(prompt);
 * -----
 * This function prompts the user for a color band, reads in a
 * color name, and returns the digit corresponding to that
 * color. If an undefined color is entered, the user is given
 * a chance to retry.
 */

static int GetColorBand(string prompt)
{
    string color;
    int digit;

    while (TRUE) {
        printf("%s", prompt);
        color = ConvertToUpperCase(GetLine());
        digit = FindStringInArray(color, colorTable, nColors);
        if (digit >= 0) break;
        printf("Illegal color -- try again.\n");
    }
    return (digit);
}

/*
 * Function: FindStringInArray
 * Usage: index = FindStringInArray(key, array, n);
 * -----
 * This function returns the index of the first element in the
 * specified array of strings that matches the value key. If
 * key does not appear in the first n elements of the array,
 * FindStringInArray returns -1.
 */

static int FindStringInArray(string key, string array[], int n)
{
    int i;

    for (i = 0; i < n; i++) {
        if (StringEqual(key, array[i])) return (i);
    }
    return (-1);
}
```

2.

```
#include <stdio.h>
#include "genlib.h"
#include "strlib.h"
#include "simpio.h"

#define MinNumber 1
#define MaxNumber 100

static string MakePrompt(string s1, int n, string s2);
static bool GetYesOrNo(string prompt);

main()
{
    int low, high, guess;
    string prompt;

    low = MinNumber;
    high = MaxNumber;
    printf("Think of a number between %d and %d", low, high);
    printf(" and I'll guess it.\n");
    while (low <= high) {
        guess = (low + high) / 2;
        prompt = MakePrompt("Is it ", guess, "? ");
        if (GetYesOrNo(prompt)) break;
        prompt = MakePrompt("Is it less than ", guess, "? ");
        if (GetYesOrNo(prompt)) {
            high = guess - 1;
        } else {
            low = guess + 1;
        }
    }
    if (low > high) {
        printf("You must have cheated. There's no such number.\n");
    } else {
        printf("I guessed the number!\n");
    }
}

/*
 * Function: MakePrompt
 * Usage: prompt = MakePrompt(prefix, value, suffix);
 * -----
 * This function creates a string consisting of the integer value
 * inserted between the prefix and suffix strings.
 */

static string MakePrompt(string s1, int n, string s2)
{
    return (Concat(s1, Concat(IntegerToString(n), s2)));
}
```

```
/*
 * Function: GetYesOrNo
 * Usage: if (GetYesOrNo(prompt)) . . .
 * -----
 * This function asks the user the question indicated by prompt
 * and waits for a yes/no response. If the user answers "yes"
 * or "no", the program returns TRUE or FALSE accordingly.
 * If the user gives any other response, the program asks
 * the question again.
 */

static bool GetYesOrNo(string prompt)
{
    string answer;

    while (TRUE) {
        printf("%s", prompt);
        answer = GetLine();
        if (StringEqual(answer, "yes")) return (TRUE);
        if (StringEqual(answer, "no")) return (FALSE);
        printf("Please answer yes or no.\n");
    }
}
```

3.

```
/*
 * Function: IsSorted
 * Usage: if (IsSorted(array, n)) . . .
 * -----
 * This function checks the first n elements of the integer
 * array to make sure they are in nondecreasing order. If
 * all the elements are correctly ordered, the function
 * returns TRUE; if any elements are out of order, it returns
 * FALSE.
 */

bool IsSorted(int array[], int n)
{
    int i;

    for (i = 1; i < n; i++) {
        if (array[i-1] > array[i]) return (FALSE);
    }
    return (TRUE);
}
```

4. The `sort.c` and `sort.h` interfaces shown here also include the function `SortRealArray`, which is required for exercise 5.

```
/*
 * File: sort.h
 * -----
 * This interface extends the sort.h interface given in the text
 * by adding both a SortRealArray and an Alphabetize function.
 */

#ifndef _sort_h
#define _sort_h

/*
 * Function: SortIntegerArray
 * Usage: SortIntegerArray(array, n);
 * -----
 * This function sorts the first n elements in array into
 * increasing numerical order. In order to use this procedure,
 * you must declare the array in the calling program and pass
 * the effective number of elements as the parameter n.
 * In most cases, the array will have a larger allocated
 * size.
 */

void SortIntegerArray(int array[], int n);

/*
 * Function: SortRealArray
 * Usage: SortRealArray(array, n);
 * -----
 * This function is identical to SortIntegerArray except that
 * the array consists of doubles instead of integers.
 */

void SortRealArray(double array[], int n);

/*
 * Function: Alphabetize
 * Usage: Alphabetize(array, n);
 * -----
 * This function sorts the first n elements in the string array
 * into increasing lexicographic order, as determined by the
 * character coding sequence, which is typically ASCII. Because
 * upper and lower case letters have different codes, "a" comes
 * after "Z" in lexicographic order.
 */

void Alphabetize(string array[], int n);

#endif
```

```
/*
 * File: sort.c
 * -----
 * This file implements the extended sort.h interface.
 */

#include <stdio.h>
#include "genlib.h"
#include "strlib.h"
#include "sort.h"

/* Private function prototypes */

static int FindSmallestInteger(int array[], int low, int high);
static void SwapIntegerElements(int array[], int p1, int p2);
static int FindSmallestReal(double array[], int low, int high);
static void SwapRealElements(double array[], int p1, int p2);
static int FindSmallestString(string array[], int low, int high);
static void SwapStringElements(string array[], int p1, int p2);

/*
 * Function: SortIntegerArray
 * -----
 * This implementation uses an algorithm called selection sort,
 * which can be described in English as follows. With your left
 * hand, point at each element in the array in turn, starting at
 * index 0. At each step in the cycle:
 *
 * (1) Find the smallest element in the range between your left
 *     hand and the end of the array, and point at that element
 *     with your right hand.
 *
 * (2) Move that element into its correct index position by
 *     switching the elements indicated by your left and right
 *     hands.
 */

void SortIntegerArray(int array[], int n)
{
    int lh, rh;

    for (lh = 0; lh < n-1; lh++) {
        rh = FindSmallestInteger(array, lh, n-1);
        SwapIntegerElements(array, lh, rh);
    }
}

/*
 * Function: SortRealArray
 * -----
 * This implementation is identical in structure to the SortIntegerArray
 * implementation. The only change is in the element type.
 */

void SortRealArray(double array[], int n)
{
    int lh, rh;

    for (lh = 0; lh < n-1; lh++) {
        rh = FindSmallestReal(array, lh, n-1);
        SwapRealElements(array, lh, rh);
    }
}
```



```
/*
 * Function: Alphabetize
 * -----
 * This implementation is again identical in structure to the
 * SortIntegerArray implementation except for the element type.
 */

void Alphabetize(string array[], int n)
{
    int lh, rh;

    for (lh = 0; lh < n-1; lh++) {
        rh = FindSmallestString(array, lh, n-1);
        SwapStringElements(array, lh, rh);
    }
}

/*
 * Function: FindSmallestInteger
 * Usage: index = FindSmallestInteger(array, low, high);
 * -----
 * This function returns the index of the smallest value in the
 * specified array of integers, searching only between the index
 * positions low and high, inclusive. It operates by keeping track
 * of the index of the smallest so far in the variable spos. If the
 * index range is empty, the function returns low.
 */

static int FindSmallestInteger(int array[], int low, int high)
{
    int i, spos;

    spos = low;
    for (i = low; i <= high; i++) {
        if (array[i] < array[spos]) spos = i;
    }
    return (spos);
}

/*
 * Function: SwapIntegerElements
 * Usage: SwapIntegerElements(array, p1, p2);
 * -----
 * This function swaps the elements in array at index
 * positions p1 and p2.
 */

static void SwapIntegerElements(int array[], int p1, int p2)
{
    int tmp;

    tmp = array[p1];
    array[p1] = array[p2];
    array[p2] = tmp;
}
```

```
/*
 * Functions: FindSmallestReal, SwapRealElements
 * -----
 * These functions are essentially the same as their integer counterparts.
 */

static int FindSmallestReal(double array[], int low, int high)
{
    int i, spos;

    spos = low;
    for (i = low; i <= high; i++) {
        if (array[i] < array[spos]) spos = i;
    }
    return (spos);
}

static void SwapRealElements(double array[], int p1, int p2)
{
    double tmp;

    tmp = array[p1];
    array[p1] = array[p2];
    array[p2] = tmp;
}

/*
 * Functions: FindSmallestString, SwapStringElements
 * -----
 * Once again, these functions are the same as their integer counterparts.
 */

static int FindSmallestString(string array[], int low, int high)
{
    int i, spos;

    spos = low;
    for (i = low; i <= high; i++) {
        if (StringCompare(array[i], array[spos]) < 0) spos = i;
    }
    return (spos);
}

static void SwapStringElements(string array[], int p1, int p2)
{
    string tmp;

    tmp = array[p1];
    array[p1] = array[p2];
    array[p2] = tmp;
}
```

5.

```
/*
 * Function: Median
 * Usage: median = Median(array, n);
 * -----
 * This function returns the median of an array of doubles.
 * The median is defined to be the central element of a sorted
 * distribution or the average of the two central elements if
 * there are an even number of values.
 */

static double Median(double array[], int n)
{
    SortRealArray(array, n);
    if (IsEven(n)) {
        return ((array[n/2] + array[n/2 + 1]) / 2);
    } else {
        return (array[n/2]);
    }
}

/*
 * Function: IsEven
 * Usage: if (IsEven(n)) . . .
 * -----
 * This function returns TRUE if n is even.
 */

static bool IsEven(int n)
{
    return (n % 2 == 0);
}
```

6. The simplest solution to the mode problem is to count every value, as demonstrated by the following implementation:

```
/*
 * Function: Mode
 * Usage: mode = Mode(array, n);
 * -----
 * This implementation just goes through the array and
 * counts the number of times each element appears, keeping
 * track of the current maximum value.
 */

static int Mode(int array[], int n)
{
    int mode, max, count, i;

    max = 0;
    for (i = 0; i < n; i++) {
        count = CountIntegerElement(array[i], array, n);
        if (count > max) {
            max = count;
            mode = array[i];
        }
    }
    return (mode);
}

/*
 * Function: CountIntegerElement
 * Usage: n = CountIntegerElement(x, array, n);
 * -----
 * This function returns the number of times the element x
 * appears in the array.
 */

static int CountIntegerElement(int value, int array[], int n)
{
    int count, i;

    count = 0;
    for (i = 0; i < n; i++) {
        if (value == array[i]) count++;
    }
    return (count);
}
```

If you sort the array first, the values to be compared are then adjacent to one another, which makes it possible to use the following alternative strategy:

```
/*
 * Function: Mode
 * Usage: mode = Mode(array, n);
 * -----
 * This implementation sorts the array and then counts
 * the length of each run of consecutive equal elements.
 * Given the efficiency of the sort function from Chapter
 * 12, this approach is not actually more efficient than
 * the count each element strategy. With a better sorting
 * algorithm, such as the one presented in Chapter 17,
 * this strategy is considerably more efficient.
 * Note that an extra test must be made at the end of the
 * for loop to ensure that the last value is checked as a
 * possible mode.
 */

static int Mode(int array[], int n)
{
    int start, count, i, max, mode;

    SortIntegerArray(array, n);
    start = max = 0;
    count = 1;
    for (i = 1; i < n; i++) {
        if (array[start] == array[i]) {
            count++;
        } else {
            if (count > max) {
                mode = array[start];
                max = count;
            }
            start = i;
            count = 1;
        }
    }
    if (count > max) mode = array[start];
    return (mode);
}
```

7.

```
/*
 * Function: RemoveZeroElements
 * Usage: n = RemoveZeroElements(array, n);
 * -----
 * This function goes through the first n elements of the array, and
 * removes any elements whose value is zero. The remaining elements
 * are shifted to fill up the cells formerly occupied by the zero
 * values, so that the final result is that the array contains only
 * the non-zero values in its initial elements. The length of that
 * new array is returned as the value of the function.
 */

static int RemoveZeroElements(int array[], int n)
{
    int lh, rh;

    lh = 0;
    for (rh = 0; rh < n; rh++) {
        if (array[rh] != 0) {
            array[lh] = array[rh];
            lh++;
        }
    }
    return (lh);
}
```

8.

```
/*
 * Function: RemoveDuplicates
 * Usage: n = RemoveDuplicates(array, n);
 * -----
 * This function removes duplicate elements from a sorted array
 * of integers and returns the new effective size. The algorithm
 * maintains two indices, lh and rh, which go through the array.
 * The lh index marks the position of the last element found. As
 * long as the element at position rh is the same, the algorithm
 * skips ahead, advancing rh. When a different element is found,
 * the lh index is advanced and the new element is copied into
 * place. Note that the algorithm must check for the empty list
 * as a special case.
 */

static int RemoveDuplicates(int array[], int n)
{
    int lh, rh;

    if (n == 0) return (n);
    lh = 0;
    for (rh = 1; rh < n; rh++) {
        if (array[lh] != array[rh]) {
            lh++;
            array[lh] = array[rh];
        }
    }
    return (lh + 1);
}
```

9.

```

/*
 * File: shuffle.c
 * -----
 * This program tests the ShuffleIntegerArray function.
 */

#include <stdio.h>
#include "genlib.h"
#include "random.h"

/*
 * Constants
 * -----
 * MinValue -- Smallest element in shuffled array
 * MaxValue -- Largest element in shuffled array
 */

#define MinValue 1
#define MaxValue 52

/* Private function prototypes */

static void ShuffleIntegerArray(int array[], int n);
static void PrintIntegerArray(int array[], int n);
static void SwapIntegerElements(int array[], int p1, int p2);

/* Main program */

main()
{
    int array[MaxValue - MinValue + 1];
    int i;

    printf("This program shuffles the integers between");
    printf(" %d and %d.\n", MinValue, MaxValue);
    for (i = MinValue; i <= MaxValue; i++) {
        array[i - MinValue] = i;
    }
    ShuffleIntegerArray(array, MaxValue - MinValue + 1);
    PrintIntegerArray(array, MaxValue - MinValue + 1);
}

/*
 * Function: ShuffleIntegerArray
 * Usage: ShuffleIntegerArray(array, n);
 * -----
 * This function randomly shuffles the first n elements in array.
 * The algorithm is exactly the same as that for sorting, except
 * that the new first element is chosen randomly instead of by
 * finding the largest element.
 */

static void ShuffleIntegerArray(int array[], int n)
{
    int lh, rh;

    for (lh = 0; lh < n; lh++) {
        rh = RandomInteger(lh, n - 1);
        SwapIntegerElements(array, lh, rh);
    }
}

/* PrintIntegerArray and SwapIntegerElements are given in the text. */

```

10.

```
/*
 * File: dutch.c
 * -----
 * This program solves the Dutch National Flag puzzle posed by
 * Edsger Dijkstra. In this puzzle, you start with an array
 * containing values representing three different colors: red,
 * white, and blue. The goal of the puzzle is to rearrange the
 * colors by swapping positions until the colors are arranged
 * in the order of the Dutch Flag: all the reds, followed by
 * all the whites, and finally all the blues.
 */

#include <stdio.h>
#include "genlib.h"
#include "random.h"
#include "sort.h"

/*
 * Constants
 * -----
 * NCells    -- The number of cells in the flag
 * TraceFlag -- Set this to TRUE to trace swaps
 */

#define NCells    15
#define TraceFlag TRUE

/* Private function prototypes */

static void DutchFlagPuzzle(char flag[NCells]);
static void InitColors(char flag[NCells]);
static char ChooseRandomColor(void);
static void SwapColors(char flag[NCells], int p1, int p2);
static void DisplayFlag(char flag[NCells]);
static int Min(int x, int y);
static int Max(int x, int y);

/* Main program */

main()
{
    char flag[NCells];

    InitColors(flag);
    if (TraceFlag) {
        printf("Initial state:\n");
        DisplayFlag(flag);
    }
    DutchFlagPuzzle(flag);
    if (!TraceFlag) DisplayFlag(flag);
}
```



```
/*
 * Function: DutchFlagPuzzle
 * Usage: DutchFlagPuzzle(flag);
 * -----
 * This function rearranges the characters in the array flag so
 * that all of the R's come first, followed by all of the W's,
 * followed by all of the B's. The algorithm uses the indices
 * rx, wx, and bx, which are subject to the following invariants:
 *
 *   flag[i] = 'R' for all i < rx
 *   flag[i] = 'W' for all i >= rx and i < wx
 *   flag[i] = 'B' for all i >= wx
 *
 * The algorithm continues until wx and bx cross.
 */

static void DutchFlagPuzzle(char flag[NCells])
{
    int rx, wx, bx;

    rx = wx = 0;
    bx = NCells;
    while (wx < bx) {
        switch (flag[wx]) {
            case 'R':
                SwapColors(flag, wx, rx);
                rx++;
                wx++;
                break;
            case 'W':
                wx++;
                break;
            case 'B':
                bx--;
                SwapColors(flag, wx, bx);
                break;
        }
    }
}

/*
 * Function: InitColors
 * Usage: InitColors(flag);
 * -----
 * This function initializes the character array flag to a random
 * sequence of letters representing the colors in the flag.
 */

static void InitColors(char flag[NCells])
{
    int i;

    for (i = 0; i < NCells; i++) {
        flag[i] = ChooseRandomColor();
    }
}
```

```
/*
 * Function: ChooseRandomColor
 * Usage: color = ChooseRandomColor();
 * -----
 * ChooseRandomColor returns a randomly chosen letter from the
 * set { R, W, B }, which represent the colors of the Dutch flag.
 */

static char ChooseRandomColor(void)
{
    switch (RandomInteger(1, 3)) {
        case 1: return ('R');
        case 2: return ('W');
        case 3: return ('B');
    }
}

/*
 * Function: SwapColors
 * Usage: SwapColors(flag, p1, p2);
 * -----
 * This function exchanges the characters in positions p1 and
 * p2 of the array flag. If TraceFlag is set, the function
 * also displays the results of the operation.
 */


static void SwapColors(char flag[NCells], int p1, int p2)
{
    char tmp;

    tmp = flag[p1];
    flag[p1] = flag[p2];
    flag[p2] = tmp;
    if (TraceFlag && p1 != p2) {
        printf("Swapping positions %d and %d\n", Min(p1, p2), Max(p1, p2));
        DisplayFlag(flag);
    }
}

/*
 * Function: DisplayFlag
 * Usage: DisplayFlag(flag);
 * -----
 * This function displays a copy of the current state of the
 * flag array on the console.
 */

static void DisplayFlag(char flag[NCells])
{
    int i;

    for (i = 0; i < NCells; i++) {
        printf("%c ", flag[i]);
    }
    printf("\n");
}
```



```
/*
 * Function: Min
 * Usage: min = Min(x, y);
 * -----
 * This function returns the smaller of the two integer values
 * x and y.
 */

static int Min(int x, int y)
{
    if (x < y) {
        return (x);
    } else {
        return (y);
    }
}

/*
 * Function: Max
 * Usage: max = Max(x, y);
 * -----
 * This function returns the larger of the two integer values
 * x and y.
 */

static int Max(int x, int y)
{
    if (x > y) {
        return (x);
    } else {
        return (y);
    }
}
```

```
11. /*
 * File: bsort.c
 * -----
 * This file implements the sort.h interface using the
 * bubble sorting algorithm.
 */

#include <stdio.h>
#include "genlib.h"
#include "sort.h"

/* Private function prototypes */

static void SwapIntegerElements(int array[], int p1, int p2);

/*
 * Function: SortIntegerArray
 * Implementation notes (see interface for usage)
 * -----
 * This implementation uses the bubble sort algorithm. In
 * a bubble sort, you go through the entire array looking
 * at pairs of adjacent elements. If the elements are out
 * of sequence, interchange them. Repeat this process until
 * a pass is made in which no swap operations are performed.
 */

void SortIntegerArray(int array[], int n)
{
    bool keepGoing;
    int i;

    keepGoing = TRUE;
    while (keepGoing) {
        keepGoing = FALSE;
        for (i = 0; i < n - 1; i++) {
            if (array[i] > array[i+1]) {
                SwapIntegerElements(array, i, i + 1);
                keepGoing = TRUE;
            }
        }
    }
}

/*
 * Function: SwapIntegerElements
 * Usage: SwapIntegerElements(array, p1, p2);
 * -----
 * This function swaps the elements in array at index
 * positions p1 and p2.
 */

static void SwapIntegerElements(int array[], int p1, int p2)
{
    int tmp;

    tmp = array[p1];
    array[p1] = array[p2];
    array[p2] = tmp;
}
```

12.

```
/*
 * File: isort.c
 * -----
 * This file implements the sort.h interface using the
 * insertion sort algorithm.
 */

#include <stdio.h>
#include "genlib.h"
#include "sort.h"

/*
 * Function: SortIntegerArray
 * Implementation notes (see interface for usage)
 * -----
 * This implementation uses the insertion sort algorithm.
 * In insertion sort, you go through the array looking at
 * each element in turn. On the ith cycle, you take out the
 * ith element and put it in the correct position with respect
 * to the elements to its left by shifting those elements over
 * one position until the appropriate position is located.
 * After you put the ith element in its correct place, the
 * first part of the array, up through index i, is sorted.
 * Thus, after the nth cycle, all n elements will be in the
 * correct position.
 */

void SortIntegerArray(int array[], int n)
{
    int i, j, temp;

    for (i = 0; i < n; i++) {
        temp = array[i];
        for (j = i - 1; j >= 0 && temp < array[j]; j--) {
            array[j + 1] = array[j];
        }
        array[j + 1] = temp;
    }
}
```

## Solutions for Chapter 13

### Pointers

#### Review questions

1. A pointer is a data item whose value is the address of some value in memory. An *lvalue* is an expression that refers to an internal memory location capable of storing data. The term *lvalue* comes from the fact that these expressions can appear on the left side of an assignment statement.
2. The four uses of pointers outlined in the introduction to this chapter are
  - Pointers allow you to refer to a large data structure in a compact way.
  - Pointers facilitate sharing data between different parts of a program.
  - Pointers make it possible to reserve new memory during program execution.
  - Pointers can be used to record relationships among data items.
3. `bool *flagp;`
4. The variable `p1` is a pointer to a `double`; `p2`, however, is simply a `double`.
5. In pointer assignment, an address is copied from one pointer variable to another, so the two pointers refer to the same storage. In value assignment, the contents of the memory addressed by a pointer are copied into the memory addressed by another pointer.
6. Memory diagram after `d2 = 2.71828`:

|      |         |     |
|------|---------|-----|
| 1000 | 3.14159 | d1  |
| 1008 | 2.71828 | d2  |
| 1016 | 1000    | dp1 |
| 1020 | 1008    | dp2 |

Memory diagram at the end of all operations:

|      |         |     |
|------|---------|-----|
| 1000 | 3.14159 | d1  |
| 1008 | 0.0     | d2  |
| 1016 | 1008    | dp1 |
| 1020 | 1008    | dp2 |

7. True.
8. False. The expression `&*x` is undefined if `x` is not a pointer variable.
9. The constant `NULL` is represented internally as 0.

10. On most machines, dereferencing a **NULL** pointer ends up selecting the contents of address 0, which is almost certainly not what you want.
11. The phrase *call by reference* refers to the process of passing the address of a variable to a function so that the function can manipulate the data in that variable.
12. The variable **x** is passed by reference and can therefore be affected by the call. The variable **y** is passed as a value and is therefore protected against changes.
13. Call by reference is most useful when a function needs to return multiple results.
14. To determine the value of

```
&intArray[j + 3];
```

the computer must first compute the integer value **j + 3**. That value is then multiplied by the size of an integer in bytes and added to the base address of **intArray**.

15. The selection expression **arr[2]** refers to the contents of the array element at index position 2. The expression **arr + 2** refers to the address of that element and therefore computes the same pointer value as the expression **&arr[2]**.
16. 1040
17.
  - a. **x++** has the value 2. After evaluation, **x** will be 3, and **y** will be unchanged.
  - b. **--x** has the value 1. After evaluation, **x** will be 1, and **y** will be unchanged.
  - c. **x++ + ++y** has the value 8. After evaluation, **x** will be 3, and **y** will be 6.
  - d. **y += x--** has the value 7. After evaluation, **x** will be 1, and **y** will be 7.
18. One such example is **array[x] = x++**, which is ambiguous because the calculation of the subscript **x** depends on the order of evaluation.
19. False. The expression **p++** adds the size of the base type to the internal representation of **p**.
20. In the expression **\*p++**, the **++** operator is applied first. In C, unary operators are applied from right to left.
21. False. Declaring an array allocates storage for the elements of the array, but declaring a variable as a pointer does not.
22. The heap is a collection of memory that is not initially in use but is available for allocation by the program.
23. Calling **malloc(n)** reserves *n* consecutive bytes of memory from the heap and returns the address of the first allocated byte. If there is not enough memory to allocate the requested storage, **malloc** returns **NULL**.
24. The type **void \*** is compatible with all pointer types, which makes it possible for programmers to design general functions, like **malloc**, that can be used with many base types.
25.
 

```
bool *flags;
flags = NewArray(100, bool);
```
26. The difference between **malloc** and **GetBlock** is that **GetBlock** checks for the out-of-memory error as part of its operation.
27. The function **free** returns dynamically allocated storage to the heap when the storage is no longer needed.
28. The term *garbage collection* refers to a storage allocation strategy in which the system itself detects what memory is no longer needed, making it unnecessary for clients of dynamic allocation packages to call **free** explicitly.

## Programming exercises

1.

```

/*
 * File: printptr.c
 * -----
 * This program tests the allocation strategy for global and
 * local variables by displaying addresses for variables of
 * each type.
 */

#include <stdio.h>
#include "genlib.h"

/* Global variables */

static int globalCount;
static double globalArray[100];
static char *globalPointer;

/* Main program */

main()
{
    int count;
    double array[100];
    char *cp;

    printf("Address of globalCount   = %lu\n", (long) &globalCount);
    printf("Address of globalArray   = %lu\n", (long) globalArray);
    printf("Address of globalPointer = %lu\n", (long) &globalPointer);
    printf("\n");
    printf("Address of count = %lu\n", (long) &count);
    printf("Address of array = %lu\n", (long) array);
    printf("Address of cp     = %lu\n", (long) &cp);
}

```

Running this program will produce different results depending on the machine.

2.

```

/*
 * File: getdate.c
 * -----
 * This program tests the GetDate function.
 */

#include <stdio.h>
#include "genlib.h"
#include "strlib.h"
#include "simpio.h"

/*
 * Global variable: monthAbbreviations
 * -----
 * This array keeps the abbreviations for each of the 12 months.
 */

static string monthAbbreviations[] = {
    "jan", "feb", "mar", "apr", "may", "jun",
    "jul", "aug", "sep", "oct", "nov", "dec"
};

```



```

/* Private function prototypes */

static void GetDate(int *dp, int *mp, int *yp);
static int FindStringInArray(string key, string array[], int n);

/* Main program */

main()
{
    int day, month, year;

    printf("Enter a date as dd-mmm-yy: ");
    GetDate(&day, &month, &year);
    printf("Day = %d\n", day);
    printf("Month = %d\n", month);
    printf("Year = %d\n", year);
}

/*
 * Function: GetDate
 * Usage: GetDate(&day, &month, &year);
 * -----
 * This function reads in a date from the user in the form dd-mmm-yy,
 * where dd is a one- or two-digit day of the month, mmm is a three-
 * letter abbreviation for a month, and yy is a two-digit year. The
 * three components of the date are "returned" by storing them in the
 * integer variables whose addresses are passed as arguments.
 */

static void GetDate(int *dp, int *mp, int *yp)
{
    int day, month, year;
    string line, monthName;

    line = GetLine();
    switch (StringLength(line)) {
        case 8: line = Concat("0", line); break;
        case 9: break;
        default: Error("Date format error: wrong length");
    }
    if (IthChar(line, 2) != '-' || IthChar(line, 6) != '-') {
        Error("Date format error: missing or misplaced -");
    }
    day = StringToInteger(SubString(line, 0, 1));
    monthName = ConvertToLowerCase(SubString(line, 3, 5));
    year = StringToInteger(SubString(line, 7, 8));
    month = FindStringInArray(monthName, monthAbbreviations, 12);
    if (month == -1) Error("Date format error: illegal month");
    *dp = day;
    *mp = month + 1;
    *yp = year;
}

/* FindStringInArray appears in Chapter 12. */

```

3.

```

/*
 * File: range.c
 * -----
 * This program computes the range of a distribution.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constants
 * -----
 * MaxElements -- The maximum number of elements in the array
 * Sentinel    -- Sentinel value used to indicate end of input
 */

#define MaxElements 100
#define Sentinel    -1

/* Private function prototypes */

static void Range(double array[], int n, double *lowp, double *highp);
static int GetRealArray(double array[], int max, double sentinel);

/* Main program */

main()
{
    double array[MaxElements], low, high;
    int n;

    printf("Enter the elements of the array, one per line.\n");
    printf("Use %g to signal the end of the list.\n", (double) Sentinel);
    n = GetRealArray(array, MaxElements, Sentinel);
    Range(array, n, &low, &high);
    printf("The range of values is %g-%g\n", low, high);
}

/*
 * Function: Range
 * Usage: Range(array, n, &low, &high);
 * -----
 * This function returns the range of a distribution by setting the
 * values of the double variables whose addresses are passed as the
 * last two parameters.
 */

static void Range(double array[], int n, double *lowp, double *highp)
{
    int i;

    if (n <= 0) Error("Range is undefined for an empty array");
    *lowp = *highp = array[0];
    for (i = 1; i < n; i++) {
        if (array[i] < *lowp) *lowp = array[i];
        if (array[i] > *highp) *highp = array[i];
    }
}

/* GetRealArray appears in the solutions for Chapter 11. */

```

4.

```
/*
 * File: sort.h
 * -----
 * This file provides an interface to a simple procedure
 * for sorting an integer array into increasing order.
 */

#ifndef _sort_h
#define _sort_h

/*
 * Function: SortIntegerArray
 * Usage: SortIntegerArray(array, n);
 * -----
 * This function sorts the first n elements in array into
 * increasing numerical order. In order to use this procedure,
 * you must declare the array in the calling program and pass
 * the effective number of elements as the parameter n.
 * In most cases, the array will have a larger allocated
 * size.
 */

void SortIntegerArray(int array[], int n);

#endif
```

```
/*
 * File: sort.c
 * -----
 * This file implements the sort.h interface using the selection
 * sort algorithm. This implementation differs from the one given
 * in Chapter 12 only in that all array references have been
 * converted to pointer references.
 */

#include <stdio.h>
#include "genlib.h"
#include "sort.h"

/* Private function prototypes */

static int *PointerToSmallestInteger(int *array, int n);
static void SwapIntegers(int *ip1, int *ip2);

/*
 * Function: SortIntegerArray
 * -----
 * This implementation uses the selection sort algorithm, recoded to
 * use pointers.
 */


void SortIntegerArray(int *array, int n)
{
    int *lh, *rh, *end;

    end = array + n;
    for (lh = array; lh < end; lh++) {
        rh = PointerToSmallestInteger(lh, end - lh);
        SwapIntegers(lh, rh);
    }
}

/*
 * Function: PointerToSmallestInteger
 * Usage: ptr = PointerToSmallestInteger(array, n);
 * -----
 * This function returns a pointer to the smallest value in the
 * specified array of integers, which has an effective size of n.
 */

static int *PointerToSmallestInteger(int *array, int n)
{
    int *ip, *sp;

    sp = array;
    for (ip = sp + 1; ip < array + n; ip++) {
        if (*ip < *sp) sp = ip;
    }
    return (sp);
}
```



```
/*
 * Function: SwapIntegers
 * Usage: SwapIntegers(ip1, ip2);
 * -----
 * This function swaps the integers whose addresses are given
 * in ip1 and ip2.
 */

static void SwapIntegers(int *ip1, int *ip2)
{
    int tmp;

    tmp = *ip1;
    *ip1 = *ip2;
    *ip2 = tmp;
}
```

5.

```
/*
 * Function: IndexArray
 * Usage: array = IndexArray(n);
 * -----
 * This function returns a dynamically allocated array of n
 * integers, each of which is initialized to its own index.
 */

static int *IndexArray(int n)
{
    int i, *array;

    array = NewArray(n, int);
    for (i = 0; i < n; i++) {
        array[i] = i;
    }
    return (array);
}
```

6.

```
/*
 * File: tabulate.c
 * -----
 * This program tabulates an array of integers.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constants
 * -----
 * MaxElements -- The maximum number of elements in the array
 * Sentinel    -- Sentinel value used to indicate end of input
 */

#define MaxElements 100
#define Sentinel    -1

/* Private function prototypes */

static void Range(int array[], int n, int *lowp, int *highp);
static void Tabulate(int array[], int n);
static int GetIntegerArray(int array[], int max, int sentinel);

/* Main program */

main()
{
    int array[MaxElements], low, high;
    int n;

    printf("Enter the elements of the array, one per line.\n");
    printf("Use %d to signal the end of list.\n", Sentinel);
    n = GetIntegerArray(array, MaxElements, Sentinel);
    Tabulate(array, n);
}

static void Tabulate(int array[], int n)
{
    int i, low, high;
    int *counts;

    Range(array, n, &low, &high);
    counts = GetBlock((high - low + 1) * sizeof(int));
    for (i = low; i <= high; i++) {
        counts[i - low] = 0;
    }
    for (i = 0; i < n; i++) {
        counts[array[i] - low]++;
    }
    for (i = low; i <= high; i++) {
        if (counts[i - low] > 0) {
            printf("%3d:%2d\n", i, counts[i - low]);
        }
    }
}
```

```

/*
 * Function: Range
 * Usage: Range(array, n, &low, &high);
 * -----
 * This function returns the range of a distribution by setting the
 * values of the integer variables whose addresses are passed as the
 * last two parameters.
 */

static void Range(int array[], int n, int *lowp, int *highp)
{
    int i;

    if (n <= 0) Error("Range is undefined for an empty array");
    *lowp = *highp = array[0];
    for (i = 1; i < n; i++) {
        if (array[i] < *lowp) *lowp = array[i];
        if (array[i] > *highp) *highp = array[i];
    }
}

/* GetIntegerArray is defined in Chapter 11. */

```

7.

```

/*
 * File: myalloc.h
 * -----
 * This interface exports a very simple dynamic allocation
 * facility.
 */

#ifndef _myalloc_h
#define _myalloc_h

#include "genlib.h"

/*
 * Function: MyGetBlock(nBytes)
 * Usage: ptr = MyGetBlock(nBytes);
 * -----
 * This function returns a pointer to a newly allocated block
 * of memory containing nBytes of storage and is therefore
 * functionally equivalent to GetBlock. If no memory is
 * available, MyGetBlock generates an error. With this
 * allocator, it is not possible to free previously allocated
 * memory.
 */

void *MyGetBlock(int nBytes);

#endif

```

```
/*
 * File: myalloc.c
 * -----
 * This file implements the myalloc.h interface. The allocation
 * strategy is extremely simple and consists of parcelling out
 * space from a static array.
 */

#include <stdio.h>
#include "genlib.h"
#include "myalloc.h"

/*
 * Constants
 * -----
 * HeapSize -- Number of bytes available in the heap
 */

#define HeapSize 1000

/*
 * Global variables
 * -----
 * myHeap    -- This array holds the available storage
 * nextPtr   -- Points to the next free block
 * bytesLeft -- Number of bytes remaining
 */

static char myHeap[HeapSize];
static char *nextPtr = myHeap;
static int bytesLeft = HeapSize;

/* Exported entries */

void *MyGetBlock(int nBytes)
{
    void *result;

    if (nBytes > bytesLeft) Error("No memory available.");
    result = nextPtr;
    nextPtr += nBytes;
    bytesLeft -= nBytes;
    return (result);
}
```



8.

```
/*
 * File: declare.c
 * -----
 * This program reads variable declarations and indicates how
 * many bytes are allocated to them. The input lines must fit
 * the form:
 *
 *      <type> <specifier-list>;
 *
 * where <type> is one a built-in type (char, int, short, long,
 * float, or double) and <specifier-list> is a list of individual
 * variable specifiers separated by commas. Each variable
 * specifier must fit one of the following forms:
 *
 *      <name>
 *      * <name>
 *      <name> [ <integer> ]
 *      * <name> [ <integer> ]
 *
 */

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

#include "genlib.h"
#include "strlib.h"
#include "simpio.h"
#include "scanner.h"

/* Private function declarations */

static void ParseDeclarationLine(string line);
static int TypeSize(string name);

/* Main program */


main()
{
    string line, token;

    IgnoreSpaces(TRUE);
    printf("Enter variable declarations, ending with a blank line.\n");
    while (TRUE) {
        line = GetLine();
        if (StringEqual(line, "")) break;
        ParseDeclarationLine(line);
    }
}
```

```
/*
 * Function: ParseDeclarationLine
 * Usage: ParseDeclarationLine(line);
 * -----
 * This function processes a declaration line and displays a message
 * indicating how much storage is allocated to each variable.
 */

static void ParseDeclarationLine(string line)
{
    string type, token, name;
    int size, nBytes, nElements, addr;
    bool isPointer;

    InitScanner(line);
    type = GetNextToken();
    size = TypeSize(type);
    while (TRUE) {
        isPointer = FALSE;
        nElements = 1;
        token = GetNextToken();
        if (StringEqual(token, "**")) {
            isPointer = TRUE;
            token = GetNextToken();
        }
        name = token;
        token = GetNextToken();
        if (StringEqual(token, "[")) {
            nElements = StringToInteger(GetNextToken());
            if (!StringEqual(GetNextToken(), "]")) {
                Error("Missing ]");
            }
            token = GetNextToken();
        }
        if (isPointer) {
            nBytes = nElements * sizeof (void *);
        } else {
            nBytes = nElements * size;
        }
        printf("%s requires %d byte%s\n",
            name, nBytes, (nBytes > 1) ? "s" : "");
        if (!StringEqual(token, ",")) break;
    }
    if (!StringEqual(token, ";")) {
        Error("Found '%s' when expecting , or ;", token);
    }
}
```



```
/*
 * Function: TypeSize
 * Usage: size = TypeSize(name);
 * -----
 * This function returns the size in bytes for each of the built-in
 * names indicated as a string.
 */

static int TypeSize(string name)
{
    if (StringEqual(name, "char")) {
        return (sizeof(char));
    } else if (StringEqual(name, "int")) {
        return (sizeof(int));
    } else if (StringEqual(name, "long")) {
        return (sizeof(long));
    } else if (StringEqual(name, "short")) {
        return (sizeof(short));
    } else if (StringEqual(name, "float")) {
        return (sizeof(float));
    } else if (StringEqual(name, "double")) {
        return (sizeof(double));
    } else {
        Error("Illegal type name %s", name);
    }
}
```

## Solutions for Chapter 14

### Strings Revisited

#### Review questions

1. This chapter asks you to think about strings as arrays of characters, as pointers to characters, and as abstract units.
2. C stores a null character (`'\0'`) after the last data character in a string to mark the end.
3. The following code processes the characters in a string treated as an array:

```
for (i = 0; str[i] != '\0'; i++) {
    . . . body of loop that manipulates str[i] . . .
}
```

If the string is instead viewed as a pointer to a character, the following code has the same effect:

```
for (cp = str; *cp != '\0'; cp++)
    . . . body of loop that manipulates *cp . . .
}
```

4. True.
5. False. Declaring a local variable as a character array reserves space for the characters; declaring it as a pointer does not allocate any space beyond that needed to hold the address.
6. Arrays cannot appear on the left side of an assignment and cannot be returned as the value of a function.
7. The storage for a local array is deallocated when the function declaring it returns. If you return a pointer to that array, the caller then has a pointer to memory that is no longer available for use.
8. The principal difference between the abstraction models presented by `string.h` and `strlib.h` lies in the memory allocation discipline. When you are using the ANSI `string.h` interface, you are responsible for allocating any string storage you need. The `strlib.h` interface automatically allocates new storage as needed.
9. In the function call `strcpy(s1, s2)`, `s2` is the source and `s1` is the destination.
10. As with any function in the ANSI `string.h` interface, you must ensure that there is enough allocated memory in the destination string to hold the entire result.
11. Buffer overflow occurs when a program writes data past the end of an array.
12. The `strcpy` function copies the null character marking the end of the string only if there is room for it in the destination string, so the caller must check the length explicitly in most applications. In addition, `strcpy` is highly inefficient for short strings because its definition forces it to copy null characters into all unused positions in the destination array.
13. The test `strcmp(s1, s2) == 0` has the same effect as `StringEqual(s1, s2)`.
14. The functions `StringLength` and `StringCompare` are implemented as pass-through functions to `strlen` and `strcmp`, respectively. With these functions in the `strlib.h` interface, clients can use it to do most string operations and do not have to learn how to use the `string.h` interface as well.
15. The library implementation of `strcpy` is usually engineered to be as efficient as possible and often runs faster than the corresponding `for` loops.

## Programming exercises

1.

```
/*
 * Function: strlen
 * Usage: len = strlen(s);
 * -----
 * This function computes the length of a string by searching for
 * the null character at the end of the character array.
 */

static int strlen(char s[])
{
    int i;

    for (i = 0; s[i] != '\0'; i++);
    return (i);
}
```

2.

```
/*
 * Function: strlen
 * Usage: len = strlen(s);
 * -----
 * This function computes the length of a string by advancing a
 * pointer until it hits the null character and then subtracting
 * the starting pointer.
 */

static int strlen(char *s)
{
    char *cp;

    for (cp = s; *cp != '\0'; cp++);
    return (cp - s);
}
```

## 3. Array version:

```
/*
 * Function: strcmp
 * Usage: cmp = strcmp(s1, s2);
 * -----
 * This function compares two strings character by character.
 * The comparison proceeds as long as the two strings match or
 * until the end is reached. There are two important details
 * to note:
 *
 * (1) It is not necessary to include checks for null
 *     characters beyond the one shown here. If one
 *     string is shorter than the other, the character
 *     values at the final position will not match because
 *     one is a null and the other is a data character.
 *
 * (2) When a difference is detected, subtracting the
 *     second ASCII value from the first gives a result
 *     of the correct sign.
 *
 * (To be strictly correct, the implementation should cast
 * the character values to the type unsigned char before
 * performing the comparison.)
 */

static int strcmp(char s1[], char s2[])
{
    int i;

    for (i = 0; s1[i] == s2[i] && s1[i] != '\0'; i++);
    return (s1[i] - s2[i]);
}
```

## Pointer version:

```
static int strcmp(char *s1, char *s2)
{
    for (; *s1 != '\0' && *s1 == *s2; s1++, s2++);
    return (*s1 - *s2);
}
```

4.

```
/*
 * File: acronym.c
 * -----
 * This file reimplements the acronym program using the
 * ANSI string library. The structure changes so that
 * the program calls a GenerateAcronym procedure rather
 * than an Acronym function.
 */

#include <stdio.h>
#include <string.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constant
 * -----
 * MaxAcronym -- Maximum number of characters in acronym
 */

#define MaxAcronym 10

/* Private function prototypes */

static void GenerateAcronym(char dst[], char src[]);

/* Main program */

main()
{
    char *str;
    char acronym[MaxAcronym + 1];

    printf("This program generates acronyms.\n");
    printf("Indicate end of input with a blank line.\n");
    while (TRUE) {
        printf("String: ");
        str = GetLine();
        if (strcmp(str, "") == 0) break;
        GenerateAcronym(acronym, str);
        printf("The acronym is %s.\n", acronym);
    }
}
```

```
/*
 * Function: GenerateAcronym
 * Usage: GenerateAcronym(dst, src);
 * -----
 * This function computes the acronym of a string, where an acronym
 * is formed by taking the initial letter of each word. The input
 * is passed as the string src and is assumed to be a sequence of
 * words separated by spaces. The result is written to the string
 * buffer whose address is passed in dst. The client must allocate
 * at least MaxAcronym+1 bytes to the destination buffer.
 */

static void GenerateAcronym(char dst[], char src[])
{
    int sx, dx;
    bool wordStart;

    dx = 0;
    wordStart = TRUE;
    for (sx = 0; src[sx] != '\0'; sx++) {
        if (src[sx] == ' ') {
            wordStart = TRUE;
        } else {
            if (wordStart) {
                if (dx == MaxAcronym - 1) Error("Acronym too long");
                dst[dx++] = src[sx];
                wordStart = FALSE;
            }
        }
    }
    dst[dx] = '\0';
}
```

5.

```
/*
 * Function: ReverseString
 * Usage: ReverseString(str);
 * -----
 * This procedure reverses the characters in the string str. The
 * result is written directly into the original string, and no new
 * storage is allocated.
 */

static void ReverseString(string str)
{
    int lh, rh;
    char tmp;

    for (lh = 0, rh = strlen(str) - 1; lh < rh; lh++, rh--) {
        tmp = str[lh];
        str[lh] = str[rh];
        str[rh] = tmp;
    }
}
```



6.

```

/*
 * File: cipher.c
 * -----
 * This program reimplements the cipher exercise from Chapter 9,
 * which encodes a message by adding a fixed offset to each
 * character, cycling from the end of the alphabet back to the
 * beginning.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "genlib.h"
#include "simpio.h"

/* Private function prototypes */

static void EncodeString(char str[], int key);

/* Main program */

main()
{
    string str;
    int key;

    printf("This program encodes a message using a cyclic cipher.\n");
    printf("Enter the numeric key: ");
    key = GetInteger();
    printf("Enter a message: ");
    str = GetLine();
    EncodeString(str, key);
    printf("Encoded message: %s\n", str);
}

/*
 * Function: EncodeString
 * Usage: EncodeString(str, key);
 * -----
 * This procedure encodes the string by replacing every letter
 * in str by the letter that comes key letters further on in
 * the alphabet. The alphabet is assumed to cycle around so
 * that 'A' comes after 'Z'. This function returns its result
 * "in place," which means that the characters in str are changed
 * to their new values.
 */

static void EncodeString(char str[], int key)
{
    char ch;
    int i;

    if (key < 0) key = 26 - abs(key) % 26;
    for (i = 0; (ch = str[i]) != '\0'; i++) {
        if (islower(ch)) {
            str[i] = 'a' + ((ch - 'a') + key) % 26;
        } else if (isupper(ch)) {
            str[i] = 'A' + ((ch - 'A') + key) % 26;
        }
    }
}

```

7.

```

/*
 * Function: SubString
 * Usage: t = SubString(s, p1, p2);
 * -----
 * SubString returns a copy of the substring of s consisting
 * of the characters between index positions p1 and p2,
 * inclusive. The following special cases apply:
 *
 * 1. If p1 is less than 0, it is assumed to be 0.
 * 2. If p2 is greater than the index of the last string
 *    position, which is StringLength(s) - 1, then p2 is
 *    set equal to StringLength(s) - 1.
 * 3. If p2 < p1, SubString returns the empty string.
 */

static string SubString(string s, int p1, int p2)
{
    int len;
    string result;

    len = strlen(s);
    if (p1 < 0) p1 = 0;
    if (p2 >= len) p2 = len - 1;
    len = p2 - p1 + 1;
    if (len <= 0) return ("");
    result = CreateString(len);
    strncpy(result, s + p1, len);
    result[len] = 0;
    return (result);
}

/*
 * Function: CreateString
 * Usage: s = CreateString(len);
 * -----
 * This function dynamically allocates space for a string of
 * len characters, leaving room for the null character at the
 * end.
 */

static string CreateString(int len)
{
    return ((string) GetBlock(len + 1));
}

```

8.

```
/*
 * Function: FindString
 * Usage: p = FindString(s, text, start);
 * -----
 * Beginning at position start in the string text, this
 * function searches for the string s and returns the
 * first index at which it appears, or -1 if no match is
 * found.
 */

static int FindString(string s, string text, int start)
{
    char *cp;

    if (start > strlen(text)) return (-1);
    if (start < 0) start = 0;
    cp = strstr(text + start, s);
    if (cp == NULL) return (-1);
    return (cp - text);
}
```

9.

```
/*
 * Function: ConvertToUpperCase
 * Usage: s = ConvertToUpperCase(s);
 * -----
 * This function returns a new string with all
 * alphabetic characters converted to upper case.
 */

string ConvertToUpperCase(string s)
{
    string result;
    int i;

    result = CreateString(strlen(s));
    for (i = 0; s[i] != '\0'; i++) result[i] = toupper(s[i]);
    result[i] = '\0';
    return (result);
}

/*
 * Function: CreateString
 * Usage: s = CreateString(len);
 * -----
 * This function dynamically allocates space for a string of
 * len characters, leaving room for the null character at the
 * end.
 */

static string CreateString(int len)
{
    return ((string) GetBlock(len + 1));
}
```

10.

```

/*
 * File: scanner.h
 * -----
 * This file redefines the scanner interface from Chapter 10
 * so that it fits more closely the structure of the ANSI string
 * libraries. As in the original package, a token is defined
 * to be either
 *
 * 1. a string of consecutive letters and digits representing
 *    a word, or
 *
 * 2. a one-character string representing a separator
 *    character, such as a space or a punctuation mark.
 *
 * To use this package, you must first call
 *
 *     InitScanner(line);
 *
 * where line is the string (typically a line returned by
 * GetLine) that is to be divided into tokens. To retrieve
 * each token in turn, you call
 *
 *     GetNextToken(buffer, bufferSize);
 *
 * where buffer is a character array allocated by the caller
 * and bufferSize is the size of that buffer.
 *
 * After the last token has been read, the predicate function
 * AtEndOfLine returns TRUE, so that the loop structure
 *
 *     while (!AtEndOfLine()) {
 *         GetNextToken(buffer, bufferSize);
 *         . . . process the token . . .
 *     }
 *
 * serves as an idiom for processing each token on the line.
 *
 * Further details for each function are given in the
 * individual descriptions below.
 */

#ifndef _scanner_h
#define _scanner_h

#include "genlib.h"

/*
 * Function: InitScanner
 * Usage: InitScanner(line);
 * -----
 * This function initializes the scanner and sets it up so that
 * it reads tokens from line. After InitScanner has been called,
 * the first call to GetNextToken will return the first token
 * on the line, the next call will return the second token,
 * and so on.
 */

void InitScanner(string line);

```

```
/*
 * Function: GetNextToken
 * Usage: GetNextToken(buffer, bufferSize);
 * -----
 * This function returns the next token on the line. If the size
 * of the token exceeds the specified size, an error is generated.
 */

void GetNextToken(char buffer[], int bufferSize);

/*
 * Function: AtEndOfLine
 * Usage: if (AtEndOfLine()) . . .
 * -----
 * This function returns TRUE when the scanner has reached
 * the end of the line.
 */

bool AtEndOfLine(void);

#endif
```

```
/*
 * File: scanner.c
 * -----
 * This file implements the scanner.h interface.
 */

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

#include "genlib.h"
#include "scanner.h"

/*
 * Private variables
 * -----
 * lineBuffer -- Private copy of the string passed to InitScanner
 * lineLength -- Length of the buffer, saved for efficiency
 * cpos       -- Current character position in the buffer
 */

static string lineBuffer;
static int lineLength;
static int cpos;

/*
 * Function: InitScanner
 * -----
 * All this function has to do is initialize the private
 * variables used in the package.
 */

void InitScanner(string line)
{
    lineBuffer = line;
    lineLength = strlen(lineBuffer);
    cpos = 0;
}
```

```
/*
 * Function: GetNextToken
 * -----
 * The implementation of GetNextToken follows its behavioral
 * description as given in the interface: if the next character
 * is alphanumeric (i.e., a letter or digit), the function
 * searches to find an unbroken string of such characters and
 * returns the entire string. If the current character is not
 * a letter or digit, a one-character string containing that
 * character is returned.
 */

void GetNextToken(char buffer[], int bufferSize)
{
    char ch;
    int bp;

    if (cpos >= lineLength) Error("No more tokens");
    bp = 0;
    ch = lineBuffer[cpos];
    if (isalnum(ch)) {
        while (cpos < lineLength && isalnum(lineBuffer[cpos])) {
            if (bp == bufferSize - 1) Error("Token too long");
            buffer[bp++] = lineBuffer[cpos++];
        }
    } else {
        if (bp == bufferSize - 1) Error("Token too long");
        buffer[bp++] = lineBuffer[cpos++];
    }
    buffer[bp] = '\0';
}

/*
 * Function: AtEndOfLine
 * -----
 * This implementation compares the current buffer position
 * against the saved length.
 */

bool AtEndOfLine(void)
{
    return (cpos >= lineLength);
}
```

11.

```
/*
 * File: piglatin.c
 * -----
 * This program translates a line of text from English
 * to Pig Latin. The rules for forming Pig Latin words
 * are as follows:
 *
 * o If the word begins with a vowel, add "way" to the
 *   end of the word.
 *
 * o If the word begins with a consonant, extract the set
 *   of consonants up to the first vowel, move that set
 *   of consonants to the end of the word, and add "ay".
 */

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

#include "genlib.h"
#include "simpio.h"
#include "scanner.h"

/*
 * Constants
 * -----
 * MaxWord -- Maximum length of an English word
 */

#define MaxWord 20

/* Private function prototypes */

static void TranslateLine(char line[]);
static bool IsLegalWord(char token[]);
static void TranslateWord(char english[], char piglatin[]);
static int FindFirstVowel(char word[]);
static bool IsVowel(char ch);

/* Main program */

main()
{
    string line;

    printf("Enter a line: ");
    line = GetLine();
    TranslateLine(line);
}
```

```
/*
 * Function: TranslateLine
 * Usage: TranslateLine(line);
 * -----
 * This function takes a line of text and translates
 * the words in the line to Pig Latin, displaying the
 * translation as it goes.
 */

static void TranslateLine(char line[])
{
    char english[MaxWord + 1];
    char piglatin[MaxWord + 4];

    InitScanner(line);
    while (!AtEndOfLine()) {
        GetNextToken(english, MaxWord + 1);
        if (IsLegalWord(english)) {
            TranslateWord(english, piglatin);
            printf("%s", piglatin);
        } else {
            printf("%s", english);
        }
    }
    printf("\n");
}

/*
 * Function: IsLegalWord
 * Usage: if (IsLegalWord(token)) . . .
 * -----
 * IsLegalWord returns TRUE if every character in the argument
 * token is alphabetic.
 */

static bool IsLegalWord(char token[])
{
    int i;

    for (i = 0; token[i] != '\0'; i++) {
        if (!isalpha(token[i])) return (FALSE);
    }
    return (TRUE);
}
```



```
/*
 * Function: TranslateWord
 * Usage: TranslateWord(english, piglatin);
 * -----
 * This function translates a word from English to Pig Latin
 * and returns the translated word.
 */

static void TranslateWord(char english[], char piglatin[])
{
    int vp;

    vp = FindFirstVowel(english);
    if (vp == -1) {
        strcpy(piglatin, english);
    } else if (vp == 0) {
        strcpy(piglatin, english);
        strcat(piglatin, "way");
    } else {
        strcpy(piglatin, &english[vp]);
        strncat(piglatin, english, vp);
        strcat(piglatin, "ay");
    }
}

/*
 * Function: FindFirstVowel
 * Usage: k = FindFirstVowel(word);
 * -----
 * FindFirstVowel returns the index position of the first vowel
 * in word. If word does not contain a vowel, FindFirstVowel
 * returns -1.
 */

static int FindFirstVowel(char word[])
{
    int i;

    for (i = 0; word[i] != '\0'; i++) {
        if (IsVowel(word[i])) return (i);
    }
    return (-1);
}

/* The IsVowel function is given in Chapter 9. */
```

12.

```

/*
 * File: hangman.c
 * -----
 * This program plays a game of hangman. This implementation uses
 * the ANSI string.h library.
 */

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

#include "genlib.h"
#include "simpio.h"
#include "random.h"

/*
 * Constant
 * -----
 * MaxWordLen -- Maximum word length
 * NGuesses   -- Number of guesses allowed
 */

#define MaxWordLen 20
#define NGuesses 8

/*
 * Global variables: wordList, nWords
 * -----
 * The array wordList contains the list of words that the computer
 * can choose for the hangman game; the size of the array is stored
 * in the variable nWords. To add new words to the game, just add
 * them to the list of static initializers below. A better solution
 * is to read this word list from a data file, but this approach
 * depends on material in Chapter 15.
 */

static char *wordList[] = {
    "ABSTRACT", "ALFALFA", "AMBASSADOR", "BOOK", "COMPUTER",
    "CRYPT", "DISPEL", "FOYER", "FUZZY", "GNOME", "GYPSY",
    "HUBBUB", "IMP", "JUNK", "KEYHOLE", "MESSY", "ONYX",
    "PUEBLO", "QUAGMIRE", "QUIET", "SLITHER", "SQUIRMY",
    "SUCCUMB", "TELEVISION", "WALLABY", "ZIRCON",
};

static int nWords = sizeof wordList / sizeof wordList[0];

/* Private function prototypes */

static void GiveInstructions(void);
static void PlayHangmanGame(void);
static char ReadUsersGuess(void);
static void InitializePattern(char pattern[], int len);
static bool UpdatePattern(char ch, char pattern[], char secret[]);

/* Main program */

main()
{
    Randomize();
    GiveInstructions();
    PlayHangmanGame();
}

```

```
/*
 * Function: GiveInstructions
 * Usage: GiveInstructions();
 * -----
 * This function prints the rules to Hangman.
 */

static void GiveInstructions(void)
{
    printf("Let's play hangman! I will pick a secret word.\n");
    printf("On each turn, you guess a letter. If the letter\n");
    printf("is in the secret word, I will show you where it\n");
    printf("appears. If you make an incorrect guess, part of\n");
    printf("your body gets strung up on the scaffold. The\n");
    printf("object is to guess the word before you are hanged.\n");
}

/*
 * Function: PlayHangmanGame
 * Usage: PlayHangmanGame();
 * -----
 * This function plays one game of Hangman with the user. The
 * program first chooses a secret word from the list stored in
 * the dictionary and then creates a pattern word of the same
 * length using only hyphens. As the game proceeds, the pattern
 * characters are replaced by the actual letters in the secret
 * word. If the word is guessed or the player exceeds the guess
 * limit, the game is over.
 */

static void PlayHangmanGame(void)
{
    char *secret, pattern[MaxWordLen+1], guess;
    int guessesLeft;

    secret = wordList[RandomInteger(0, nWords - 1)];
    InitializePattern(pattern, strlen(secret));
    guessesLeft = NGuesses;
    while (guessesLeft > 0 && strcmp(pattern, secret) != 0) {
        printf("The word now looks like this: %s\n", pattern);
        if (guessesLeft == 1) {
            printf("You have only one guess left.\n");
        } else {
            printf("You have %d guesses left.\n", guessesLeft);
        }
        guess = ReadUsersGuess();
        if (UpdatePattern(guess, pattern, secret)) {
            printf("That guess is correct.\n");
        } else {
            printf("There are no %c's in the word.\n", guess);
            guessesLeft--;
        }
    }
    if (guessesLeft == 0) {
        printf("You've run out of guesses.\n");
        printf("The word was: %s\n", secret);
        printf("You lose.\n");
    } else {
        printf("You guessed the word: %s\n", secret);
        printf("You win.\n");
    }
}
```

```
/*
 * Function: ReadUserGuess
 * Usage: ch = ReadUserGuess();
 * -----
 * This function requests a guess from the user and checks to see
 * if it is legal.  Legal guesses are single letters, which are
 * converted to uppercase before being returned.  If a guess is
 * not legal, the user is asked to supply a new guess.
 */


static char ReadUsersGuess(void)
{
    char guess, *line;

    while (TRUE) {
        printf("Your guess: ");
        line = GetLine();
        if (strlen(line) != 1) {
            printf("Your guess must be a single character.");
        } else if (!isalpha(guess = line[0])) {
            printf("Your guess must be a letter.");
        } else {
            return (toupper(guess));
        }
        printf(" Try again.\n");
    }
}

/*
 * Function: InitializePattern
 * Usage: InitializePattern(pattern, len);
 * -----
 * This function initializes the pattern string to be a sequence
 * of hyphens.  The length of the pattern is given by len.
 */

static void InitializePattern(char pattern[], int len)
{
    int i;

    for (i = 0; i < len; i++) {
        pattern[i] = '-';
    }
    pattern[len] = '\0';
}
```



```
/*
 * Function: UpdatePattern
 * Usage: if (UpdatePattern(ch, pattern, secret)) . . .
 * -----
 * This function goes through the pattern string and updates
 * it so that it contains the same characters except for
 * those positions in which ch occurs in the secret word.
 * UpdatePattern returns TRUE if any updates are made.
 */

static bool UpdatePattern(char ch, char pattern[], char secret[])
{
    int i;
    bool result;

    result = FALSE;
    for (i = 0; pattern[i] != '\0'; i++) {
        if (secret[i] == ch) {
            pattern[i] = ch;
            result = TRUE;
        }
    }
    return (result);
}
```

## Solutions for Chapter 15

### Files

#### Review questions

1. A file is represented internally as a one-dimensional sequence of characters.
2. The process of opening a file consists of associating a file name with an internal file handle represented as a pointer to a **FILE** structure.
3. The type **FILE \*** is used to hold the information the system needs to keep track of an open data file. In most cases, the details of the underlying **FILE** structure are relevant only to the implementors of the file system and not to programmers who are simply using files as part of their application.
4. The second argument to **fopen** represents the mode in which the file is opened. The strings "**r**", "**w**", and "**a**" indicate reading from a file, writing to a file, and appending to a file, respectively.
5. The **fopen** function reports failure by returning the pointer value **NULL**.
6. Closing files explicitly is good programming practice and makes it easier for other programmers to embed your code in new applications.
7. The **stdio.h** interface defines the standard files **stdin**, **stdout**, and **stderr**, which refer to the standard input file, the standard output file, and the standard error-reporting file, respectively. In most environments, all three of these files refer to the console by default.
8. False. The function **getc** returns a value of type **int**.
9. The **getc** function returns the special sentinel value **EOF** to indicate the end of the input file.
10. The usual strategy for updating a file involves the following six steps:
  - Step 1. Open the original file for input.
  - Step 2. Open a temporary file for output with a different name.
  - Step 3. Copy the input file to the temporary file, making any required editing changes in the process.
  - Step 4. Close both files.
  - Step 5. Delete the original file.
  - Step 6. Rename the temporary file so that it once again has the original name.
11. If you call **rename(f1, f2)**, the file that used to be called **f1** is given the new name **f2**.
12. The function **ungetc** returns a character to an input stream so that it can be read again by the next call to one of the input functions. The advantage of **ungetc** is that using it often simplifies the structure of an application by allowing the caller to, in essence, look ahead at the next character to decide what to do, without having to modify the subsequent processing to take account of the fact that the first character in a sequence has already been read.
13. The typical call is **fgets(buffer, bufSize, infile)**, where **buffer** is a character array large enough to hold the input line, **bufSize** is the allocated size of the buffer, and **infile** is the input file pointer.
14. The major differences between **fgets** and **ReadLine** are that
  - **ReadLine** allocates its own heap memory as needed, making it impossible to overflow the buffer.
  - **ReadLine** removes the newline character used to signal the end of the line, so that the string returned to the caller consists only of the actual characters on the line.
  - Each string returned by **ReadLine** is stored in its own memory, so that no confusion can occur as to whether a string needs to be copied before it is stored.
15. The **printf** function always directs its output to the standard output device. The **fprintf** function has the same effect except that the output goes to a user-specified file pointer. The **sprintf** function writes its output to a string buffer.
16. True.

17. False. The arguments to **scanf** that indicate the destinations for the various values must be pointers, but need not be specified explicitly using an ampersand. For string conversions in particular, the usual approach is to declare a character array and then use the array name—without an ampersand—in the **scanf** call. The array name is defined to be a pointer to its first element and therefore satisfies the rule that the argument must be a pointer.
18. White-space characters are those for which the **isspace** predicate in **ctype.h** returns **TRUE**.
19. A white-space character in a **scanf** control string instructs **scanf** to skip over any white-space characters in the input.
20. The specification **%f** is incompatible with the type of the variable **d**. The correct **scanf** call is  

```
scanf("%d, %lf", &i, &d);
```
21. The **scanf** specification **"%10[^, :]"** reads a string of up to 10 characters terminated by the first occurrence of a comma or colon.
22. False. As developer P. J. Plaugher reports, the “usefulness [of **scanf**], over the years, has proved to be limited.”

## Programming exercises

1.

```
/*
 * File: wc.c
 * -----
 * This program reports the number of lines, words, and
 * characters in an input file. The name "wc" is too short
 * to be much help to someone trying to understand the purpose
 * of this program but is used because that is the name for
 * this utility program on Unix.
 */

#include <stdio.h>
#include <ctype.h>
#include "genlib.h"
#include "simpio.h"

/* Function prototypes */

static FILE *OpenUserFile(string prompt, string mode);

/* Main program */

main()
{
    int lines, words, chars;
    int ch;
    bool inword;
    FILE *infile;

    lines = words = chars = 0;
    inword = FALSE;
    infile = OpenUserFile("File: ", "r");
    while ((ch = getc(infile)) != EOF) {
        chars++;
        if (ch == '\n') lines++;
        if (isspace(ch)) {
            if (inword) {
                words++;
                inword = FALSE;
            }
        } else {
            inword = TRUE;
        }
    }
    fclose(infile);
    if (inword) words++;
    printf("Lines: %4d\n", lines);
    printf("Words: %4d\n", words);
    printf("Chars: %4d\n", chars);
}

/* The OpenUserFile function appears in the text. */
```



2.

```
/*
 * File: greek.c
 * -----
 * This program displays a file with all letters converted to
 * X or x depending on the case of the original letter.
 */

#include <stdio.h>
#include <ctype.h>
#include "genlib.h"
#include "simpio.h"
#include "random.h"

/* Private function prototypes */

static FILE *OpenUserFile(string prompt, string mode);

/* Main program */

main()
{
    FILE *infile;
    int ch;

    infile = OpenUserFile("Old file: ", "r");
    while ((ch = getc(infile)) != EOF) {
        if (isupper(ch)) {
            ch = RandomInteger('A', 'Z');
        } else if (islower(ch)) {
            ch = RandomInteger('a', 'z');
        }
        putchar(ch);
    }
    fclose(infile);
}

/* The OpenUserFile function appears in the text. */
```

3.

```
/*
 * File: untabify.c
 * -----
 * This program replaces tabs in a file with the appropriate
 * number of spaces. To update the original file, this
 * program first copies the original to a temporary file
 * without any editing and then copies data back to the
 * original, removing tabs as it goes.
 */

#include <stdio.h>
#include <ctype.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constants
 * -----
 * TabColumns -- Number of columns for each tab stop
 */

#define TabColumns 8

/* Private function prototypes */

static void UntabifyFile(FILE *infile, FILE *outfile);

/* Main program */

main()
{
    string filename, temp;
    FILE *infile, *outfile;

    printf("This program converts tabs into multiple spaces.\n");
    while (TRUE) {
        printf("File name: ");
        filename = GetLine();
        infile = fopen(filename, "r");
        if (infile != NULL) break;
        printf("File %s not found -- try again.\n", filename);
    }
    temp = tmpnam(NULL);
    outfile = fopen(temp, "w");
    if (outfile == NULL) Error("Can't open temporary file");
    UntabifyFile(infile, outfile);
    fclose(infile);
    fclose(outfile);
    if (remove(filename) != 0 || rename(temp, filename) != 0) {
        Error("Unable to rename temporary file");
    }
}
```

```
/*
 * Function: UntabifyFile
 * Usage: UntabifyFile(infile, outfile);
 * -----
 * This function copies data from infile to outfile, converting
 * tabs into a sequence of spaces. Tabs always generate at least
 * one space, after which they generate enough spaces to reach the
 * next tab stop.
 */

static void UntabifyFile(FILE *infile, FILE *outfile)
{
    int column, ch;

    column = 0;
    while ((ch = getc(infile)) != EOF) {
        if (ch == '\t') {
            putc(' ', outfile);
            column++;
            while (column % TabColumns != 0) {
                putc(' ', outfile);
                column++;
            }
        } else {
            if (ch == '\n') {
                column = 0;
            } else {
                column++;
            }
            putc(ch, outfile);
        }
    }
}
```

4.

```
/*
 * File: ufile.c
 * -----
 * This program copies one file to another converting strings
 * surrounded by asterisks into underlined words.
 */

#include <stdio.h>
#include <ctype.h>
#include "genlib.h"
#include "simpio.h"

/* Private function prototypes */

static void UnderlineFile(FILE *infile, FILE *outfile);
static FILE *OpenUserFile(string prompt, string mode);

/* Main program */

main()
{
    FILE *infile, *outfile;

    printf("This program copies one file to another, underlining\n");
    printf("words surrounded by asterisks.\n");
    infile = OpenUserFile("Old file: ", "r");
    outfile = OpenUserFile("New file: ", "w");
    UnderlineFile(infile, outfile);
    fclose(infile);
    fclose(outfile);
}
```

```
/*
 * Function: UnderlineFile
 * Usage: UnderlineFile(infile, outfile);
 * -----
 * This function copies data from infile to outfile, converting
 * strings surrounded by asterisks into underlined words by
 * preceding each character in the range with an underscore
 * followed by a backspace. Newline characters in the string are
 * never underlined, and an asterisk followed by a white-space
 * character is considered to be a real asterisk and not the start
 * of an underlining sequence.
 *
 * The program uses the flag underlining to mark whether it is
 * in underlining mode. The program enters underlining mode when
 * it sees an asterisk not followed by white space. It leaves
 * underlining mode on any asterisk. For normal characters, the
 * program simply displays the underlining sequence before the
 * character.
 */

static void UnderlineFile(FILE *infile, FILE *outfile)
{
    bool underlining;
    int ch;

    underlining = FALSE;
    while ((ch = getc(infile)) != EOF) {
        if (ch == '*') {
            if (underlining) {
                underlining = FALSE;
            } else {
                ch = getc(infile);
                ungetc(ch, infile);
                if (isspace(ch)) {
                    putc('*', outfile);
                } else {
                    underlining = TRUE;
                }
            }
        } else {
            if (underlining && ch != '\n') {
                fprintf(outfile, "_\b");
            }
            putc(ch, outfile);
        }
    }
}

/* The OpenUserFile function appears in the text. */
```

5.

```
/*
 * File: rev.c
 * -----
 * This program reverses a file using ReadFile to read
 * the entire file at once.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"
#include "readfile.h"

/* Private function prototypes */

static int StringArrayLength(string array[]);
static FILE *OpenUserFile(string prompt, string mode);

/* Main program */

main()
{
    FILE *infile;
    string *lines;
    int i;

    printf("This program reverses a file and displays the result.\n");
    infile = OpenUserFile("Old file: ", "r");
    lines = ReadFile(infile);
    fclose(infile);
    for (i = StringArrayLength(lines) - 1; i >= 0; i--) {
        printf("%s\n", lines[i]);
    }
}

/*
 * Function: StringArrayLength
 * Usage: n = StringArrayLength(array);
 * -----
 * This function returns the length of a NULL-terminated string
 * array.
 */

static int StringArrayLength(string array[])
{
    int i;

    for (i = 0; array[i] != NULL; i++);
    return (i);
}

/* The OpenUserFile function appears in the text. */
```

```
/*
 * File: readfile.h
 * -----
 * This interface exports a function that reads in an entire
 * file at once, returning the result as a dynamic string array.
 */

#ifndef _readfile_h
#define _readfile_h

#include "genlib.h"

/*
 * Function: ReadFile
 * Usage: lines = ReadFile(infile);
 * -----
 * This function takes an open file pointer and reads the entire
 * file as a sequence of lines. The lines are stored in a dynamic
 * string array, which is returned as the value of the function.
 * The end of the data is marked by a NULL entry in the array.
 */

string *ReadFile(FILE *infile);

#endif
```

```
/*
 * File: readfile.c
 * -----
 * This file implements the readfile.h interface.
 */

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

#include "genlib.h"
#include "strlib.h"
#include "simpio.h"
#include "readfile.h"

/*
 * Constant: InitialBufferSize
 * -----
 * This constant defines the initial buffer size used in the
 * ReadFile function. In some sense the value is arbitrary,
 * since ReadFile allocates new memory as needed.
 */

#define InitialBufferSize 256

/* Exported entries */

string *ReadFile(FILE *infile)
{
    string *buffer, *nbuffer, line;
    int i, n, size;

    n = 0;
    size = InitialBufferSize;
    buffer = NewArray(size, string);
    while ((line = ReadLine(infile)) != NULL) {
        if (n == size) {
            size *= 2;
            nbuffer = NewArray(size, string);
            for (i = 0; i < n; i++) {
                nbuffer[i] = buffer[i];
            }
            FreeBlock(buffer);
            buffer = nbuffer;
        }
        buffer[n++] = line;
    }
    nbuffer = NewArray(n + 1, string);
    for (i = 0; i < n; i++) {
        nbuffer[i] = buffer[i];
    }
    nbuffer[n] = NULL;
    FreeBlock(buffer);
    return (nbuffer);
}
```



6.

```
/*
 * File: madlibs.c
 * -----
 * This program implements the game of Madlibs, in which the
 * user is asked to supply words corresponding to particular
 * parts of speech. These words are then used to fill blanks
 * in a hidden text.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"
#include "strlib.h"

/* Private function prototypes */

static void ProcessMadlibsFile(FILE *infile, FILE *outfile);
static void ProcessLine(string line, FILE *outfile);
static void CopyFile(FILE *infile, FILE *outfile);
static FILE *OpenUserFile(string prompt, string mode);

/* Main program */

main()
{
    string temp;
    FILE *infile, *outfile;

    infile = OpenUserFile("Input file: ", "r");
    temp = tmpnam(NULL);
    outfile = fopen(temp, "w");
    if (outfile == NULL) Error("Can't open temporary file");
    ProcessMadlibsFile(infile, outfile);
    fclose(infile);
    fclose(outfile);
    infile = fopen(temp, "r");
    if (infile == NULL) Error("Can't reopen temporary file");
    printf("\n");
    CopyFile(infile, stdout);
    fclose(infile);
    remove(temp);
}

static void ProcessMadlibsFile(FILE *infile, FILE *outfile)
{
    string line;

    while ((line = ReadLine(infile)) != NULL) {
        ProcessLine(line, outfile);
    }
}
```

```
static void ProcessLine(string line, FILE *outfile)
{
    int i, start;
    bool inPlaceholder;
    string word, substitution;

    inPlaceholder = FALSE;
    for (i = 0; line[i] != '\0'; i++) {
        if (inPlaceholder) {
            if (line[i] == '>') {
                word = SubString(line, start, i - 1);
                printf("  %s: ", word);
                substitution = GetLine();
                fprintf(outfile, substitution);
                inPlaceholder = FALSE;
            }
        } else {
            if (line[i] == '<') {
                start = i + 1;
                inPlaceholder = TRUE;
            } else {
                putc(line[i], outfile);
            }
        }
    }
    if (inPlaceholder) Error("Unmatched placeholder markers");
    putc('\n', outfile);
}

/*
 * Function: CopyFile
 * Usage: CopyFile(infile, outfile);
 * -----
 * This function copies the contents of infile to outfile. The
 * client is responsible for opening these files before calling
 * CopyFile and for closing them afterward.
 */

static void CopyFile(FILE *infile, FILE *outfile)
{
    int ch;

    while ((ch = getc(infile)) != EOF) {
        putc(ch, outfile);
    }
}

/* The OpenUserFile function appears in the text. */
```

7.

```

/*
 * File: wordfreq.c
 * -----
 * This program goes through a file and keeps track of how
 * often each word appears. At the end of the program, the
 * table of words is printed in alphabetical order.
 */

#include <stdio.h>
#include <ctype.h>
#include "genlib.h"
#include "strlib.h"
#include "simpio.h"
#include "scanner.h"
#include "freqtab.h"

/* Private function prototypes */

static void RecordWordsInLine(string line);
static bool IsLegalWord(string token);
static FILE *OpenUserFile(string prompt, string mode);

/* Main program */

main()
{
    FILE *infile;
    string line;

    InitFrequencyTable();
    infile = OpenUserFile("Input file: ", "r");
    while ((line = ReadLine(infile)) != NULL) {
        RecordWordsInLine(line);
    }
    fclose(infile);
    printf("\nWord frequency table:\n");
    DisplayFrequencyTable();
}

/*
 * Function: RecordWordsInLine
 * Usage: RecordWordsInLine(line);
 * -----
 * This function calls RecordWord for each word on the input
 * line.
 */

static void RecordWordsInLine(string line)
{
    string word;

    InitScanner(line);
    while (!AtEndOfLine()) {
        word = GetNextToken();
        if (IsLegalWord(word)) {
            RecordWord(ConvertToLowerCase(word));
        }
    }
}

/* IsLegalWord appears in Chapter 10, OpenUserFile in Chapter 15. */

```

```
/*
 * File: freqtab.h
 * -----
 * This file is the interface to a package that supports a
 * table of word frequencies.
 *
 * To use this package, you must first call
 *
 *     InitFrequencyTable();
 *
 * to initialize the package. Each time you encounter a
 * word that you would like to include in the count, you
 * call
 *
 *     RecordWord(word);
 *
 * To generate a report showing the word frequencies, call
 *
 *     DisplayFrequencyTable();
 */

#ifndef _freqtab_h
#define _freqtab_h

#include "genlib.h"

/*
 * Function: InitFrequencyTable
 * Usage: InitFrequencyTable();
 * -----
 * This function initializes the frequency table and must be called
 * before other functions in this package.
 */

void InitFrequencyTable(void);

/*
 * Function: RecordWord
 * Usage: RecordWord(word);
 * -----
 * If word has not previously been seen, it is added to the table
 * with a count of 1; if it has, the count is incremented.
 */

void RecordWord(string word);

/*
 * Function: DisplayFrequencyTable
 * Usage: DisplayFrequencyTable();
 * -----
 * This function displays a frequency table, which appears in
 * alphabetical order.
 */

void DisplayFrequencyTable(void);

#endif
```

```
/*
 * File: freqtab.c
 * -----
 * This file implements the freqtab.h interface.
 */

#include <stdio.h>
#include "genlib.h"
#include "strlib.h"
#include "freqtab.h"

/*
 * Constants
 * -----
 * MaxWords -- Maximum number of words
 */

#define MaxWords 100

/*
 * Private variables
 * -----
 * wordTable -- Holds sorted list of words
 * wordCount -- Holds count of corresponding entry in wordTable
 * nWords    -- Number of words
 */

static string wordTable[MaxWords];
static int wordCount[MaxWords];
static int nWords;

/* Private function prototypes */

static void InsertNewRecord(string word, int pos);

/*
 * Function: InitFrequencyTable
 * -----
 * This function initializes the table. In this implementation, the
 * only required initialization is to set the table length to zero.
 */

void InitFrequencyTable(void)
{
    nWords = 0;
}
```

```
/*
 * Function: RecordWord
 * -----
 * The implementation of RecordWord exists in two pieces. First,
 * the function looks to see if the word already exists in the table
 * by executing the binary search algorithm used in the function
 * FindStringInSortedArray. If the string does not previously exist in
 * the table, the function adds a new entry by calling InsertNewRecord.
 * The implementation cannot call FindStringInSortedArray directly
 * without forcing InsertNewRecord to execute redundant operations
 * to locate the insertion point a second time.
 */


void RecordWord(string word)
{
    int lh, rh, mid, cmp;

    lh = 0;
    rh = nWords - 1;
    while (lh <= rh) {
        mid = (lh + rh) / 2;
        cmp = StringCompare(word, wordTable[mid]);
        if (cmp == 0) break;
        if (cmp < 0) {
            rh = mid - 1;
        } else {
            lh = mid + 1;
        }
    }
    if (lh > rh) InsertNewRecord(word, mid = lh);
    wordCount[mid]++;
}

/*
 * Function: DisplayFrequencyTable
 * -----
 * This function simply displays the table, which is always
 * kept in sorted order.
 */

void DisplayFrequencyTable(void)
{
    int i;

    for (i = 0; i < nWords; i++) {
        printf("%-10s %4d\n", wordTable[i], wordCount[i]);
    }
}
```



```
/* Private functions */

/*
 * Function: InsertNewRecord
 * Usage: InsertNewRecord(word, pos);
 * -----
 * This function inserts a new record for word at position pos.
 * The word is inserted with a count of 0, so that it is the
 * caller's responsibility to increment the count.
 */

static void InsertNewRecord(string word, int pos)
{
    int i;

    if (nWords == MaxWords) Error("Word limit exceeded");
    for (i = nWords; i > pos; i--) {
        wordTable[i] = wordTable[i - 1];
        wordCount[i] = wordCount[i - 1];
    }
    wordTable[pos] = word;
    wordCount[pos] = 0;
    nWords++;
}
```

8.

```

/*
 * File: order.c
 * -----
 * This program reads lines from a file corresponding to
 * a product order and prints out a formatted version of
 * the output.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constants
 * -----
 * MaxCatalogCode -- Maximum length of a catalog code
 * MaxProductName -- Maximum length of a product name
 */

#define MaxCatalogCode 6
#define MaxProductName 20

/* Private function prototypes */

static FILE *OpenUserFile(string prompt, string mode);

/* Main program */

main()
{
    FILE *infile;
    char catalogCode[MaxCatalogCode+1];
    char productName[MaxProductName+1];
    int nUnits;
    double unitPrice, totalPrice, grandTotal;
    char termch;
    int nscan;

    infile = OpenUserFile("Order file: ", "r");
    grandTotal = 0;
    while (TRUE) {
        nscan = fscanf(infile, "%6s %20[^/] / %d @ %lg%c",
                       catalogCode, productName, &nUnits, &unitPrice,
                       &termch);
        if (nscan == EOF) break;
        if (nscan != 5 || termch != '\n') {
            Error("Illegal line format");
        }
        totalPrice = nUnits * unitPrice;
        grandTotal += totalPrice;
        printf("%-8s %-20s %3d @ %7.2f = %7.2f\n",
               catalogCode, productName, nUnits, unitPrice, totalPrice);
    }
    fclose(infile);
    printf("-----\n");
    printf("%-46s%7.2f\n", "TOTAL", grandTotal);
}

/* The OpenUserFile function appears in the text. */

```



9.

```

/*
 * File: polygon.c
 * -----
 * This program reads lines in a file that consist of points in a
 * series of polygons and then displays them in the graphics window.
 *
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"
#include "graphics.h"

/*
 * Constants
 * -----
 * MaxLine -- Maximum length of an input line.
 */

#define MaxLine 60

/* Private function prototypes */

static void DrawLineTo(double x, double y);
static FILE *OpenUserFile(string prompt, string mode);

/* Main program */

main()
{
    FILE *infile;
    char line[MaxLine+1];
    double x, y, x0, y0, firstX, firstY;
    bool polygonStarted;
    char termch;
    int nscan;

    InitGraphics();
    infile = OpenUserFile("File containing polygon points: ", "r");
    polygonStarted = FALSE;
    while (fgets(line, MaxLine, infile) != NULL) {
        if (line[0] == '\n') {
            if (polygonStarted) {
                DrawLineTo(x0, y0);
                polygonStarted = FALSE;
            }
        } else {
            nscan = sscanf(line, "(%lg, %lg)%c", &x, &y, &termch);
            if (nscan != 3 || termch != '\n') {
                Error("Illegal format: %s", line);
            }
            if (polygonStarted) {
                DrawLineTo(x, y);
            } else {
                MovePen(x0 = x, y0 = y);
                polygonStarted = TRUE;
            }
        }
    }
    if (polygonStarted) DrawLineTo(x0, y0);
    fclose(infile);
}

/* DrawLineTo appears in Chapter 7, OpenUserFile in Chapter 15. */

```



## Solutions for Chapter 16

### Records

#### Review questions

1. A record is a collection of unordered, heterogeneous values that has conceptual integrity as a single unit; a field is any of the values that make up that collection. C programmers often refer to these concepts as *structures* and *members*, respectively.
2. True.
3. False. The types of each field in a record can be different, but need not be.
4. Before you declare a record variable, you must first define a new structure type representing the record. After doing so, you can then declare variables of that new type.
5. The `.` operator (pronounced *dot*) is used to select a field from a record.
6. True.
7. True.
8. Bob Cratchit's salary: `staff[1].salary`  
Ebenezer Scrooge's first initial: `staff[0].name[0]`
9. The only syntactic change required was adding an asterisk before the type name in the `typedef` definition.
10. The three factors that influence the decision as to whether a type should be declared as a record or as a pointer to a record are
  - The size of the record type
  - The cost of memory allocation
  - The discipline used by functions that manipulate the type
11. The `genlib.h` interface exports the `New` function to simplify dynamic allocation of records.
12. The expression

`*p.cost`

has the wrong operator precedence to achieve the desired effect. As written, the dot operator is applied first, followed by the asterisk; what you ordinarily want is to have these operators applied in the opposite order. To simplify this combined operation of dereference and selection, C provides the `->` operator, which is illustrated by the expression

`p->cost`

13. A database is a large collection of structured objects that can be maintained in permanent storage.
14. The external representation of the database allows it to be stored in the file system, which is often limited to ASCII text. For more convenient manipulation, however, you also need to design an internal representation that can be manipulated directly by the program. To design the external database, programmers must consider the limitations of the file storage medium and the extent to which the database should be readable by human readers. For the internal database, the design is driven by the types of operations the program needs to perform.
15. Having the course writer supply question numbers as part of the data file makes it much easier to revise the course database.
16. The term *data-driven design* refers to programs that control their entire operation on the basis of information from a database. Data-driven programs are usually shorter, more flexible, and easier to maintain than programs that incorporate the same information directly into the implementation.

## Programming exercises

1.

```

/*
 * File: payroll.c
 * -----
 * This program tests the functions defined for records of type
 * employeeT.
 */

#include <stdio.h>
#include "genlib.h"
#include "strlib.h"
#include "simpio.h"

/*
 * Constants
 * -----
 * FirmName      -- Name of firm
 * MaxEmployees  -- Maximum number of employees
 * FlatTaxRate   -- Rate of tax (Example: 0.05 is a 5% tax)
 * Deduction     -- Deduction for each dependent
 * MaxMemo       -- Maximum width of memo field on check
 */

#define FirmName      "Scrooge and Marley Ltd."
#define MaxEmployees  100
#define FlatTaxRate   0.25
#define Deduction     1.00
#define MaxMemo       25

/*
 * Type: employeeT
 * -----
 * This structure defines the fields for an employee.
 */

typedef struct {
    string name;
    string title;
    string ssn;
    double salary;
    int withholding;
} employeeT;

/*
 * Global variables
 * -----
 * staff          -- Array of employees
 * nEmployees     -- Number of employees
 */

static employeeT staff[MaxEmployees];
static int nEmployees;

/* Private function declarations */

static void InitEmployeeTable(void);
static void GeneratePayroll(employeeT staff[], int nEmployees);
static void WriteCheck(employeeT emp);
static double MaxF(double x, double y);

```

```
/* Main program */

main()
{
    InitEmployeeTable();
    GeneratePayroll(staff, nEmployees);
}

/*
 * Function: InitEmployeeTable
 * Usage: InitEmployeeTable();
 * -----
 * This function is not actually part of the assigned problem and
 * exists primarily to generate a test case. In a real payroll
 * application, the data would almost certainly be read from a
 * data file.
 */

static void InitEmployeeTable(void)
{
    employeeT emp;

    emp.name = "Ebenezer Scrooge";
    emp.title = "Partner";
    emp.ssnum = "271-82-8183";
    emp.salary = 250.00;
    emp.withholding = 1;
    staff[0] = emp;
    emp.name = "Bob Cratchit";
    emp.title = "Clerk";
    emp.ssnum = "314-15-9265";
    emp.salary = 15.00;
    emp.withholding = 7;
    staff[1] = emp;
    nEmployees = 2;
}

/*
 * Function: GeneratePayroll
 * Usage: GeneratePayroll(staff, nEmployees);
 * -----
 * This function generates a payroll report for each of the
 * employees in the array staff. The parameter nEmployees
 * gives the effective size of the staff array.
 */

static void GeneratePayroll(employeeT staff[], int nEmployees)
{
    int i;

    for (i = 0; i < nEmployees; i++) {
        WriteCheck(staff[i]);
    }
}
```

```

/*
 * Function: WriteCheck
 * Usage: WriteCheck(employee);
 * -----
 * This function writes a check for one employee according to the
 * data in the employee record.
 */

static void WriteCheck(employeeT emp)
{
    double gross, tax, net;
    char memo[MaxMemo];

    gross = emp.salary;
    tax = MaxF(0, FlatTaxRate * (gross - emp.withholding * Deduction));
    net = gross - tax;
    sprintf(memo, "%.2f gross - %.2f tax", gross, tax);
    printf("\n");
    printf("-----+\\n");
    printf("| %-40s          |\\n", FirmName);
    printf("|                                     |\\n");
    printf("| Pay to the order of: %-20s%8.2f |\\n", emp.name, net);
    printf("|                                     |\\n");
    printf("|                                     |\\n");
    printf("| %-30s          E. Scrooge |\\n", memo);
    printf("-----+\\n");
    printf("\\n");
}

/*
 * Function: MaxF
 * Usage: max = MaxF(x, y);
 * -----
 * This function returns the larger of the two floating-point
 * values x and y.
 */

static double MaxF(double x, double y)
{
    if (x > y) {
        return (x);
    } else {
        return (y);
    }
}

```

2. The required types are

```

/*
 * Types
 * -----
 * bookT      -- The information about a particular book
 * libraryDB -- The information about the library
 */

typedef struct {
    string title;
    string authors[MaxAuthors];
    int nAuthors;
    string subjects[MaxSubjects];
    int nSubjects;
    string catalogNumber;
    string publisher;
    int year;
    bool isCirculating;
} *bookT;

typedef struct {
    bookT collection[MaxBooks];
    int nBooks;
} *libraryDB;

```

The associated code is


```

/*
 * Function: SearchBySubject
 * Usage: SearchBySubject(libdata, subject);
 * -----
 * This function searches the library database for entries with
 * the given subject. For each matching book, the program writes
 * out the title, first author, and catalog number.
 */

static void SearchBySubject(libraryDB libdata, string subject)
{
    int i;

    for (i = 0; i < libdata->nBooks; i++) {
        if (SubjectMatches(libdata->collection[i], subject)) {
            printf("%s\n", libdata->collection[i]->title);
            printf("%s\n", libdata->collection[i]->authors[0]);
            printf("%s\n", libdata->collection[i]->catalogNumber);
            printf("\n");
        }
    }
}

```



```
/*
 * Function: SubjectMatches
 * Usage: if (SubjectMatches(book, subject)) . . .
 * -----
 * This function returns TRUE if the subject list for the book
 * matches the specified subject.
 */

static bool SubjectMatches(bookT book, string subject)
{
    int i;

    for (i = 0; i < book->nSubjects; i++) {
        if (StringEqual(book->subjects[i], subject)) {
            return (TRUE);
        }
    }
    return (FALSE);
}
```

3.

```
/*
 * Function: Midpoint
 * Usage: mid = Midpoint(p1, p2);
 * -----
 * This function returns the midpoint of the line segment between
 * points p1 and p2.
 */

static pointT Midpoint(pointT p1, pointT p2)
{
    pointT p;

    p.x = (p1.x + p2.x) / 2;
    p.y = (p1.y + p2.y) / 2;
    return (p);
}
```



4.

```
/*
 * File: ptgraph.h
 * -----
 * This interface provides an additional abstraction layer on
 * top of the graphics library that allows programs to specify
 * coordinates and displacements as points.
 */

#ifndef _ptgraph_h
#define _ptgraph_h

/*
 * Type: pointT
 * -----
 * This structure represents a point in the x/y plane.
 */

typedef struct {
    double x, y;
} pointT;

/*
 * Function: MovePenToPoint
 * Usage: MovePenToPoint(pt);
 * -----
 * This procedure moves the current point to pt.
 */

void MovePenToPoint(pointT pt);

/*
 * Function: DrawLineToPoint
 * Usage: DrawLineToPoint(pt);
 * -----
 * This procedure draws a line from the current point to pt.
 */

void DrawLineToPoint(pointT pt);

/*
 * Functions: GetWindowSize
 * Usage: ur = GetWindowSize();
 * -----
 * This function returns the coordinates of the upper-right
 * corner of the graphics window.
 */

pointT GetWindowSize(void);

/*
 * Functions: GetCurrentPosition
 * Usage: pt = GetCurrentPosition();
 * -----
 * This function returns the current pen position.
 */

pointT GetCurrentPosition(void);

#endif
```

```
/*
 * File: ptgraph.c
 * -----
 * This file implements the ptgraph.h interface as a layer
 * on top of graphics.h. For each of the exported functions,
 * the implementation is simple enough that additional comments
 * should not be necessary; for information of the behavior of
 * these functions, please see the comments in the interface.
 */

#include <stdio.h>
#include "genlib.h"
#include "graphics.h"
#include "ptgraph.h"

/* Exported entries */

void MovePenToPoint(pointT pt)
{
    MovePen(pt.x, pt.y);
}

void DrawLineToPoint(pointT pt)
{
    DrawLine(pt.x - GetCurrentX(), pt.y - GetCurrentY());
}

pointT GetWindowSize(void)
{
    pointT ur;

    ur.x = GetWindowWidth();
    ur.y = GetWindowHeight();
    return (ur);
}

pointT GetCurrentPosition(void)
{
    pointT cp;

    cp.x = GetCurrentX();
    cp.y = GetCurrentY();
    return (cp);
}
```

5.

```

/*
 * File: rational.h
 * -----
 * This interface provides simple functions for manipulating rational
 * numbers, which are represented as the quotient of two integers.
 */

#ifndef _rational_h
#define _rational_h

/*
 * Type: rationalT
 * -----
 * This structure represents a rational number.
 */

typedef struct {
    int num, den;
} rationalT;

/*
 * Function: CreateRational
 * Usage: rat = CreateRational(num, den);
 * -----
 * This function creates a new rational number from its components.
 * The internal representation is always reduced to lowest terms.
 */

rationalT CreateRational(int num, int den);

/*
 * Functions: AddRational, MultiplyRational
 * Usage: r = AddRational(r1, r2);
 *        r = MultiplyRational(r1, r2);
 * -----
 * These functions add and multiply rational numbers, respectively.
 */

rationalT AddRational(rationalT r1, rationalT r2);
rationalT MultiplyRational(rationalT r1, rationalT r2);

/*
 * Function: PrintRational
 * Usage: PrintRational(rat);
 * -----
 * This function displays the value of a rational number on the standard
 * output device in the form num/den. Any surrounding punctuation or
 * newline characters are the responsibility of the client.
 */

void PrintRational(rationalT rat);

/*
 * Function: GetRational
 * Usage: rat = GetRational();
 * -----
 * This function reads in a rational number from the user. The input
 * may be either a simple integer or the pair num/den.
 */

rationalT GetRational(void);

#endif

```

```
/*
 * File: rational.c
 * -----
 * This file implements the rational.h interface.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"
#include "rational.h"

/* Private function prototypes */

static rationalT ReduceToLowestTerms(rationalT old);
static int GCD(int x, int y);

/* Exported entries */

rationalT CreateRational(int num, int den)
{
    rationalT rat;

    rat.num = num;
    rat.den = den;
    return (ReduceToLowestTerms(rat));
}

rationalT AddRational(rationalT r1, rationalT r2)
{
    rationalT rat;

    rat.num = r1.num * r2.den + r2.num * r1.den;
    rat.den = r1.den * r2.den;
    return (ReduceToLowestTerms(rat));
}

rationalT MultiplyRational(rationalT r1, rationalT r2)
{
    rationalT rat;

    rat.num = r1.num * r2.num;
    rat.den = r1.den * r2.den;
    return (ReduceToLowestTerms(rat));
}

void PrintRational(rationalT rat)
{
    if (rat.den == 1) {
        printf("%d", rat.num);
    } else {
        printf("%d/%d", rat.num, rat.den);
    }
}
```

```

/*
 * Implementation notes: GetRational
 * -----
 * The GetRational function must accept a rational number in either
 * of two forms: a simple integer (indicating a denominator of 0) or
 * a pair of integers separated by a slash. This implementation uses
 * the scanf function to read the input data. Note that the white
 * space characters between the format codes means that arbitrary
 * white space can occur between the input tokens.
 */

rationalT GetRational(void)
{
    string line;
    rationalT rat;
    int nscan;
    char sep, extra;

    while (TRUE) {
        line = GetLine();
        nscan = sscanf(line, "%d %c %d %c",
                       &rat.num, &sep, &rat.den, &extra);
        FreeBlock(line);
        if (nscan == 1) {
            rat.den = 1;
            return (rat);
        } else if (nscan == 3 && sep == '/') {
            return (ReduceToLowestTerms(rat));
        }
        printf("Illegal rational number. Try again.\n");
    }
}

/* Private functions */

/*
 * Function: ReduceToLowestTerms
 * Usage: new = ReduceToLowestTerms(old);
 * -----
 * This function reduces a rational number to lowest terms by
 * dividing both numerator and denominator by their greatest
 * common divisor.
 */

static rationalT ReduceToLowestTerms(rationalT old)
{
    rationalT new;
    int gcd;

    gcd = GCD(old.num, old.den);
    new.num = old.num / gcd;
    new.den = old.den / gcd;
    return (new);
}

/* The GCD function appears in Chapter 6. */

```

```
/*
 * File: addlist.c
 * -----
 * This program adds a list of rational numbers. The end of
 * the input is indicated by entering 0 as a sentinel value.
 */

#include <stdio.h>
#include "genlib.h"
#include "rational.h"

main()
{
    rationalT value, total;

    printf("This program adds a list of rational numbers.\n");
    printf("Signal end of list with a 0.\n");
    total = CreateRational(0, 1);
    while (TRUE) {
        printf(" ? ");
        value = GetRational();
        if (value.num == 0) break;
        total = AddRational(total, value);
    }
    printf("The total is ");
    PrintRational(total);
    printf("\n");
}
```

6.

```
/*
 * File: roman.c
 * -----
 * This program converts a Roman numeral to its decimal
 * equivalent.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"
#include "strlib.h"

/*
 * Type: romanEntryT
 * -----
 * This type is used to store entries in the Roman numeral
 * table, where each entry consists of a character and its
 * corresponding integer value.
 */

typedef struct {
    char symbol;
    int value;
} romanEntryT;

/*
 * Static global data: romanTable
 * -----
 * This table contains the entire list of Roman numeral/value
 * pairs.
 */

static romanEntryT romanTable[] = {
    {'I', 1},
    {'V', 5},
    {'X', 10},
    {'L', 50},
    {'C', 100},
    {'D', 500},
    {'M', 1000}
};

static int nSymbols = sizeof romanTable / sizeof romanTable[0];

/* Private function prototypes */

static int RomanToDecimal(string str);
static int RomanLetterValue(char ch);

main()
{
    string str;

    printf("This program converts Roman numerals to decimal.\n");
    printf("Enter a blank line to stop.\n");
    while (TRUE) {
        printf("Roman numeral: ");
        str = GetLine();
        if (StringEqual(str, "")) break;
        printf("Decimal value: %d\n", RomanToDecimal(str));
    }
}
```

```
/*
 * Function: RomanToDecimal
 * Usage: decimal = RomanToDecimal(roman);
 * -----
 * This function converts a Roman numeral string into its decimal
 * equivalent, taking account of both additive ("VI") and subtractive
 * ("IV") relationships. If an illegal character appears in the
 * string, the function returns -1.
 *
 * Implementation note: This function contains a subtle dependency on
 * the fact that RomanLetterValue returns -1 if the character is not
 * in the table of legal values. In determining whether to subtract
 * or add the value to the total, the function checks the value of the
 * following letter, which will be the null character at the end of the
 * string. Because the value -1 returned by RomanLetterValue('\0') is
 * always less than the current value, the final character will always
 * be added to the total.
 */

static int RomanToDecimal(string roman)
{
    int i, total, value;

    total = 0;
    for (i = 0; roman[i] != '\0'; i++) {
        value = RomanLetterValue(roman[i]);
        if (value == -1) return (-1);
        if (value < RomanLetterValue(roman[i+1])) {
            total -= value;
        } else {
            total += value;
        }
    }
    return (total);
}

/*
 * Function: RomanLetterValue
 * Usage: value = RomanLetterValue(ch);
 * -----
 * This function finds the value of a single Roman numeral character
 * by looking it up in the internal table. If the character does not
 * appear in the table, RomanLetterValue returns -1.
 */

static int RomanLetterValue(char ch)
{
    int i;

    for (i = 0; i < 7; i++) {
        if (romanTable[i].symbol == ch) return (romanTable[i].value);
    }
    return (-1);
}
```



7.

```
/*
 * File: exchange.c
 * -----
 * This program uses a table of exchange rates stored as
 * a data file to convert from one currency unit to another.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"
#include "strlib.h"

/*
 * Constants
 * -----
 * MaxCurrencies    -- Maximum number of currencies
 * MaxCurrencyName  -- Maximum length of currency name
 * DatabaseFile     -- File name containing the currency database
 */

#define MaxCurrencies 100
#define MaxCurrencyName 25
#define DatabaseFile "exchange.dat"

/*
 * Types
 * -----
 * currencyEntryT    -- Entry for single currency
 * exchangeDatabaseT -- Information for the exchange database
 */

typedef struct {
    string name;
    double dollars;
} currencyEntryT;

typedef struct {
    currencyEntryT info[MaxCurrencies];
    int nCurrencies;
} *exchangeDatabaseT;

/* Private function prototypes */

static exchangeDatabaseT ReadExchangeRates(void);
static int FindCurrency(exchangeDatabaseT db, string name);
```

```

/* Main program */

main()
{
    exchangeDatabaseT exchangeRates;
    string oldCurrency, newCurrency;
    int oldIndex, newIndex;
    double oldValue, dollars, newValue;


    exchangeRates = ReadExchangeRates();
    printf("Convert from: ");
    oldCurrency = GetLine();
    oldIndex = FindCurrency(exchangeRates, oldCurrency);
    if (oldIndex == -1) Error("No such currency");
    printf("Into: ");
    newCurrency = GetLine();
    newIndex = FindCurrency(exchangeRates, newCurrency);
    if (newIndex == -1) Error("No such currency");
    printf("How many units of type %s? ", oldCurrency);
    oldValue = GetReal();
    dollars = oldValue * exchangeRates->info[oldIndex].dollars;
    newValue = dollars / exchangeRates->info[newIndex].dollars;
    printf("%g %s = %g %s\n", oldValue, oldCurrency,
           newValue, newCurrency);
}

/*
 * Function: ReadExchangeRates
 * Usage: db = ReadExchangeRates();
 * -----
 * This function reads the standard exchange database file into
 * the internal data structure and returns that structure.
 */

static exchangeDatabaseT ReadExchangeRates(void)
{
    exchangeDatabaseT db;
    FILE *infile;
    char namebuf[MaxCurrencyName], termch;
    int n, nscan;
    double rate;

    db = New(exchangeDatabaseT);
    infile = fopen(DatabaseFile, "r");
    if (infile == NULL) Error("No such file");
    n = 0;
    while (TRUE) {
        nscan = fscanf(infile, "%24s%lg%c",
                       namebuf, &rate, &termch);
        if (nscan == EOF) break;
        if (nscan != 3 || termch != '\n') {
            Error("Illegal format in database file");
        }
        if (n == MaxCurrencies) Error("Too many currencies");
        db->info[n].name = ConvertToLowerCase(namebuf);
        db->info[n].dollars = rate;
        n++;
    }
    fclose(infile);
    db->nCurrencies = n;
    return (db);
}

```



```
/*
 * Function: FindCurrency
 * Usage: index = FindCurrency(db, name);
 * -----
 * This function looks up the name of the currency in the database
 * and returns the index of that entry. If there is no currency
 * with the indicated name, FindCurrency returns NULL.
 */

static int FindCurrency(exchangeDatabaseT db, string name)
{
    int i;

    name = ConvertToLowerCase(name);
    for (i = 0; i < db->nCurrencies; i++) {
        if (StringEqual(db->info[i].name, name)) return (i);
    }
    return (-1);
}
```

8.

```
/*
 * File: dict.c
 * -----
 * This file implements the dict.h interface using an array of
 * records to hold the name/value pairs.
 */

#include <stdio.h>
#include "genlib.h"
#include "strlib.h"
#include "dict.h"

/*
 * Constants
 * -----
 * MaxDictSize -- Maximum number of dictionary entries
 */

#define MaxDictSize 100

/*
 * Type: dictEntryT
 * -----
 * This type holds a single entry in the dictionary.  Each
 * entry consists of a key and value pair.
 */

typedef struct {
    string key, value;
} dictEntryT;

/*
 * Private variables
 * -----
 * dictionary -- The array of dictionary entries
 * nEntries    -- Number of entries currently in the dictionary
 */

static dictEntryT dictionary[MaxDictSize];
static int nEntries;

/* Private function prototypes */

static int FindEntry(string word);

/* Exported entries */

void InitDictionary(void)
{
    nEntries = 0;
}
```

```
void Define(string word, string definition)
{
    int entry;

    entry = FindEntry(word);
    if (entry == -1) {
        if (nEntries == MaxDictSize) Error("Define: Too many words");
        entry = nEntries++;
        dictionary[entry].key = CopyString(word);
    }
    dictionary[entry].value = CopyString(definition);
}

string Lookup(string word)
{
    int entry;

    entry = FindEntry(word);
    if (entry == -1) return (NULL);
    return (dictionary[entry].value);
}

/* Private functions */

/*
 * Function: FindEntry
 * Usage: entry = FindEntry(word);
 * -----
 * This function searches through the internal dictionary to find
 * the specified word. If the word exists, FindEntry returns the
 * index in the dictionary array at which it appears. If the word
 * is not in the dictionary, FindEntry returns -1.
 */

static int FindEntry(string word)
{
    int i;

    for (i = 0; i < nEntries; i++) {
        if (StringEqual(word, dictionary[i].key)) return (i);
    }
    return (-1);
}
```

9.

```

/*
 * File: formlet.c
 * -----
 * This program generates form letters. It is essentially the
 * same program used for the game of Madlibs, except that a
 * dictionary has been added to remember previous substitutions.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"
#include "strlib.h"
#include "dict.h"

/* Private function prototypes */

static void ProcessFormLetter(FILE *infile, FILE *outfile);
static void ProcessLine(string line, FILE *outfile);
static string GetSubstitution(string word);
static void CopyFile(FILE *infile, FILE *outfile);
static FILE *OpenUserFile(string prompt, string mode);

/* Main program */

main()
{
    string temp;
    FILE *infile, *outfile;

    infile = OpenUserFile("Input file: ", "r");
    temp = tmpnam(NULL);
    outfile = fopen(temp, "w");
    if (outfile == NULL) Error("Can't open temporary file");
    ProcessFormLetter(infile, outfile);
    fclose(infile);
    fclose(outfile);
    infile = fopen(temp, "r");
    if (infile == NULL) Error("Can't reopen temporary file");
    printf("\n");
    CopyFile(infile, stdout);
    fclose(infile);
    remove(temp);
}

/*
 * Function: ProcessFormLetter
 * Usage: ProcessFormLetter(infile, outfile);
 * -----
 * This function reads form letter data from the file specified by infile
 * and writes an update letter to outfile, substituting any placeholders
 * in the text marked by <name> with a value supplied by the user. In
 * this implementation, the value associated with each name is stored in
 * a dictionary and reused if that same placeholder appears again.
 */

static void ProcessFormLetter(FILE *infile, FILE *outfile)
{
    string line;

    while ((line = ReadLine(infile)) != NULL) {
        ProcessLine(line, outfile);
    }
}

```

```

/*
 * Function: ProcessLine
 * Usage: ProcessLine(line, outfile);
 * -----
 * This function processes the characters in the string line and
 * writes the updated data to outfile, performing the substitutions
 * described for ProcessFormLetter. The implementation processes
 * the line character by character and uses the flag inPlaceholder
 * to record whether the current character is part of the text or
 * within a placeholder name.
 */

static void ProcessLine(string line, FILE *outfile)
{
    int i, start;
    bool inPlaceholder;
    string word, substitution;

    inPlaceholder = FALSE;
    for (i = 0; line[i] != '\0'; i++) {
        if (inPlaceholder) {
            if (line[i] == '>') {
                word = SubString(line, start, i - 1);
                fprintf(outfile, GetSubstitution(word));
                inPlaceholder = FALSE;
            }
        } else {
            if (line[i] == '<') {
                start = i + 1;
                inPlaceholder = TRUE;
            } else {
                putc(line[i], outfile);
            }
        }
    }
    if (inPlaceholder) Error("Unmatched placeholder markers");
    putc('\n', outfile);
}

/*
 * Function: GetSubstitution
 * Usage: sub = GetSubstitution(word);
 * -----
 * This function first tries to look up the value of word in the
 * internal dictionary. If it has been previously defined, the
 * function returns the old definition. If not, it asks the user
 * to supply a new definition for that placeholder name.
 */

static string GetSubstitution(string word)
{
    string definition;

    definition = Lookup(word);
    if (definition == NULL) {
        printf(" %s: ", word);
        definition = GetLine();
        Define(word, definition);
    }
    return (definition);
}

```

```

/*
 * Function: CopyFile
 * Usage: CopyFile(infile, outfile);
 * -----
 * This function copies the contents of infile to outfile. The
 * client is responsible for opening these files before calling
 * CopyFile and for closing them afterward.
 */

static void CopyFile(FILE *infile, FILE *outfile)
{
    int ch;

    while ((ch = getc(infile)) != EOF) {
        putc(ch, outfile);
    }
}

/* OpenUserFile is define in Chapter 15. */

```

10. There are many ways to solve this problem. One approach, for example, is to associate each of the answers with a response text for that answer. The problem with this approach is that it adds quite a bit of additional mechanism to the driver. An alternative strategy that changes the driver program very little is to allow the course writer to specify **AUTO** as the sole answer to a question. When the course processing program encounters such a question, it displays the question text and then immediately moves to the specified next question field following the **AUTO** keyword. For example, suppose that the course data file contains the following sequence:

```

2
Integer review.
True or false: The largest value of type int on the
Macintosh is 32767.
-----
true:  6
t:     6
false: 3
f:     3

3
You are incorrect. The maximum int value is 32767.
The reason is that an object of type int is represented
on the Mac using 16 bits (binary digits). Half of
the numbers are negative, and we only therefore can
run from 0 to 32767 (2 to the 15th minus 1).
-----
AUTO:  4

4
How many values are there of type char?
a.  26
b. 100
c. 128
d. 256
-----

```

Given this data file, answering question 2 incorrectly advances the course program to question 3, which displays the appropriate response. From there, however, the program automatically advances to question 4. In this design, an unexpected answer to question 4 no longer causes the previous explanation to be repeated.

Besides the comments, the only changes required to implement this new design are in the **ProcessCourse** function:



```
static void ProcessCourse(courseDB course)
{
    questionT q;
    int qnum;
    string ans;
    int index;

    printf("%s\n", course->title);
    qnum = 1;
    while (qnum != 0) {
        q = course->questions[qnum];
        AskQuestion(q);
        if (StringEqual(q->answers[0].ans, "AUTO")) {
            index = 0;
        } else {
            ans = ConvertToUpperCase(GetLine());
            index = FindAnswer(ans, q);
        }
        if (index == -1) {
            printf("I don't understand that.\n");
        } else {
            qnum = q->answers[index].nextq;
        }
    }
    printf("Finished\n");
}
```

## Solutions for Chapter 17

### Looking Ahead

#### Review questions

1. Recursion is defined as any solution technique in which large problems are solved by reducing them to smaller problems of the same form.
2. The essential difference between recursion and stepwise refinement is that recursion requires the subproblems to have the same form as the original problem.
3. The recursive leap of faith consists of believing that recursive instances of a functional call always work, even though the coding of the function itself is not yet complete.
4. The first keyword in the body of a recursive function is usually `if`.
5. The second call to `ExchangeCharacters` is necessary to ensure that the conditions for the next cycle of the loop are preserved. If you leave this step out, some permutations will be listed twice and others will disappear when you run the program on a string with more than three characters.
6. A wrapper is an extremely simple function—usually just one line—that implements one function in terms of another, often with slightly different arguments. Wrappers are useful in recursive programs because recursive functions are often not precisely the same as those exported by an interface. By defining the interface functions as wrappers that call private recursive functions with the right argument structure, you can avoid changing the interface structure.
7. The fundamental operations on a queue are `Enqueue` and `Dequeue`.
8. `typedef struct pictureCDT *pictureADT;`
9. The definition of a concrete type belongs in the implementation.
10. A ring buffer is an array in which the elements at the end are interpreted as being wrapped around back to the beginning. Using a ring buffer to represent a queue makes it easier to implement the queue as an array without wasting storage as the queue grows and shrinks.
11. The two most common simplifications you can use in big-O notation are that
  - You can throw away any terms that become insignificant as  $N$  becomes large.
  - You can ignore any constant factors.
12. The computational complexity of the selection sort algorithm is  $O(N^2)$  while the complexity of merge sort is  $O(N \log N)$ . These figures mean that if you double the number of values being sorted, the time required to perform a selection sort increases by a factor of 4, whereas the time for merge sort is doubled, plus a little bit extra. As the size of the array to be sorted becomes larger, the advantage of merge sort becomes increasingly pronounced.
13. The computational complexity of the binary-search algorithm is  $O(\log N)$ .

## Programming exercises

1.

```
/*
 * Function: Fib
 * Usage: t = Fib(n);
 * -----
 * This function returns the nth term in the Fibonacci sequence
 * using a recursive implementation of the recurrence relation
 *
 *      Fib(n) = Fib(n - 1) + Fib(n - 2)
 */

static int Fib(int n)
{
    if (n < 2) {
        return (n);
    } else {
        return (Fib(n - 1) + Fib(n - 2));
    }
}
```

2.

```
/*
 * Function: Combinations
 * Usage: ways = Combinations(n, k);
 * -----
 * This function is a recursive implementation of the Combinations
 * function, which returns the number of distinct ways of choosing
 * k objects from a set of n objects. The recursive implementation
 * is based on the structure of Pascal's triangle
 *
 *              1
 *             1 1
 *            1 2 1
 *           1 3 3 1
 *          1 4 6 4 1
 *
 * where every entry is the sum of the two entries above it. The
 * entries in the triangle are C(n, k), where n is the row number
 * and k is the index of the element in a particular row.
 */

int Combinations(int n, int k)
{
    if (k == 0 || k == n) {
        return (1);
    } else {
        return (Combinations(n - 1, k - 1) + Combinations(n - 1, k));
    }
}
```

3.

```
/*
 * Function: IsPalindrome
 * Usage: if (IsPalindrome(str)) . . .
 * -----
 * This function returns TRUE if the string is a palindrome.
 * This implementation operates recursively by noting that all
 * strings of length 0 or 1 are palindromes (the simple case)
 * and that longer strings are palindromes only if their first
 * and last characters match and the remaining substring is a
 * palindrome.
 */

bool IsPalindrome(string str)
{
    int len;

    len = StringLength(str);
    if (len <= 1) {
        return (TRUE);
    } else {
        return (str[0] == str[len - 1]
                && IsPalindrome(SubString(str, 1, len - 2)));
    }
}
```

4.

```

/*
 * File: hanoi.c
 * -----
 * This program solves the Tower of Hanoi problem.
 */

#include <stdio.h>
#include "genlib.h"

/*
 * Constants
 * -----
 * NDisks -- Number of disks in the original tower
 */

#define NDisks 3

/* Function prototypes */

static void MoveTower(int n, char start, char finish, char temp);
static void MoveSingleDisk(char start, char finish);

/* Main program */

main()
{
    MoveTower(NDisks, 'A', 'B', 'C');
}

/*
 * Function: MoveTower
 * Usage: MoveTower(n, start, finish, temp);
 * -----
 * This function is the heart of the recursive solution to the Tower
 * of Hanoi problem. Calling MoveTower(n, start, finish, temp)
 * corresponds to the English command "Move a tower of size n from
 * start to finish using temp for intermediate storage."
 */

static void MoveTower(int n, char start, char finish, char temp)
{
    if (n == 1) {
        MoveSingleDisk(start, finish);
    } else {
        MoveTower(n - 1, start, temp, finish);
        MoveSingleDisk(start, finish);
        MoveTower(n - 1, temp, finish, start);
    }
}

/*
 * Function: MoveSingleDisk
 * Usage: MoveSingleDisk(start, finish);
 * -----
 * This function encapsulates the operation of moving a single disk.
 * In this implementation, the function simply traces the movement.
 */

static void MoveSingleDisk(char start, char finish)
{
    printf("%c -> %c\n", start, finish);
}

```

5.

```
/*
 * File: fractal.c
 * -----
 * This program draws a Koch fractal.
 */

#include <stdio.h>
#include <math.h>
#include "genlib.h"
#include "simpio.h"
#include "graphics.h"

/* Private function prototypes */

static void KochFractal(double size, int order);
static void DrawFractalLine(double len, double theta, int order);
static void DrawPolarLine(double r, double theta);

/* Main program */

main()
{
    double size;
    int order;

    InitGraphics();
    printf("Program to draw Koch fractals\n");
    printf("Enter edge length in inches: ");
    size = GetReal();
    printf("Enter order of fractal: ");
    order = GetInteger();
    KochFractal(size, order);
}

/*
 * Function: KochFractal
 * Usage: KochFractal(size, order);
 * -----
 * This function draws a Koch fractal snowflake centered in
 * the graphics window of the indicated size and order.
 */

static void KochFractal(double size, int order)
{
    double x0, y0;

    x0 = GetWindowWidth() / 2 - size / 2;
    y0 = GetWindowHeight() / 2 - sqrt(3) * size / 6;
    MovePen(x0, y0);
    DrawFractalLine(size, 0, order);
    DrawFractalLine(size, 120, order);
    DrawFractalLine(size, 240, order);
}
```

```
/*
 * Function: DrawFractalLine
 * Usage: DrawFractalLine(len, theta, order);
 * -----
 * This function draws a fractal line of the given length starting
 * from the current point and moving in direction theta. If order
 * is 0, the fractal line is just a straight line. If order is
 * greater than zero, the line is divided into four line segments,
 * each of which is a fractal line of the next lower order. The
 * four segments connect the same endpoints as the straight line,
 * but include a triangular wedge replacing the center third of
 * the segment.
 */

static void DrawFractalLine(double len, double theta, int order)
{
    if (order == 0) {
        DrawPolarLine(len, theta);
    } else {
        DrawFractalLine(len/3, theta, order - 1);
        DrawFractalLine(len/3, theta - 60, order - 1);
        DrawFractalLine(len/3, theta + 60, order - 1);
        DrawFractalLine(len/3, theta, order - 1);
    }
}

/* DrawPolarLine is given in the text. */
```

6.

```
/*
 * File: menomonic.c
 * -----
 * This program generates mnemonics from a telephone number.
 */

#include <stdio.h>
#include "genlib.h"
#include "strlib.h"
#include "simpio.h"
#include "fill.h"

/*
 * Constants
 * -----
 * FillMargin -- Margin at which to fold each line
 */

#define FillMargin 47

/* Private function declarations */

static void ListMnemonics(string str);
static void RecursiveMnemonics(string prefix, string rest);
static string DigitLetters(char ch);

/* Main program */

main()
{
    string str;

    SetFillMargin(FillMargin);
    printf("This program displays mnemonics for a telephone number.\n");
    printf("Number: ");
    str = GetLine();
    ListMnemonics(str);
    PrintFilledString("\n");
}

/*
 * Function: ListMnemonics
 * Usage: ListMnemonics(str);
 * -----
 * This function lists all of the mnemonics for the string of digits
 * stored in the string str. The correspondence between digits and
 * letters is the same as that on the standard telephone dial. The
 * implementation at this level is a simple wrapper function that
 * provides the arguments necessary for the recursive call.
 */

static void ListMnemonics(string str)
{
    RecursiveMnemonics("", str);
}
```



```

/*
 * Function: RecursiveMnemonics
 * Usage: RecursiveMnemonics(prefix, rest);
 * -----
 * This function does all of the real work for ListMnemonics and
 * implements a more general problem with a recursive solution
 * that is easier to see. The call to RecursiveMnemonics generates
 * all mnemonics for the digits in the string rest prefixed by the
 * mnemonic string in prefix. As the recursion proceeds, the rest
 * string gets shorter and the prefix string gets longer.
 */

static void RecursiveMnemonics(string prefix, string rest)
{
    string head, tail, options;
    int i;

    if (StringLength(rest) == 0) {
        PrintFilledString(prefix);
        PrintFilledString(" ");
    } else {
        options = DigitLetters(IthChar(rest, 0));
        tail = SubString(rest, 1, StringLength(rest) - 1);
        for (i = 0; i < StringLength(options); i++) {
            head = Concat(prefix, CharToString(IthChar(options, i)));
            RecursiveMnemonics(head, tail);
        }
    }
}

/*
 * Function: DigitLetters
 * Usage: digits = DigitLetters(ch);
 * -----
 * This function returns a string consisting of the legal substitutions
 * for a given digit character.
 */

static string DigitLetters(char ch)
{
    switch (ch) {
        case '0': return ("0");
        case '1': return ("1");
        case '2': return ("ABC");
        case '3': return ("DEF");
        case '4': return ("GHI");
        case '5': return ("JKL");
        case '6': return ("MNO");
        case '7': return ("PRS");
        case '8': return ("TUV");
        case '9': return ("WXY");
        default: Error("Illegal digit");
    }
}

```

7.

```
/*
 * File: queue.c
 * -----
 * This file implements the queue.h abstraction using a
 * ring buffer.
 */

#include <stdio.h>

#include "genlib.h"
#include "queue.h"

/*
 * Constants:
 * -----
 * MaxQueueSize -- Maximum number of elements in the queue
 * QueueArraySize -- Size of the internal array
 */

#define MaxQueueSize 10
#define QueueArraySize (MaxQueueSize + 1)

/*
 * Type: queueCDT
 * -----
 * This type provides the concrete counterpart to the queueADT.
 * This implementation uses a ring buffer to implement the
 * queue. The next item to be dequeued is found at the array
 * element indexed by head. The tail index indicates the next
 * free position. When head and tail are equal, the queue is
 * empty. The head and tail indices each move from the end of
 * the array back to the beginning, giving rise to the name
 * "ring buffer."
 */

struct queueCDT {
    void *array[QueueArraySize];
    int head;
    int tail;
};

/* Exported entries */

/*
 * Function: NewQueue
 * -----
 * This function allocates and initializes the storage for a
 * new queue.
 */

queueADT NewQueue(void)
{
    queueADT queue;

    queue = New(queueADT);
    queue->head = queue->tail = 0;
    return (queue);
}
```

```
/*
 * Function: FreeQueue
 * -----
 * This function simply frees the queue storage.
 */

void FreeQueue(queueADT queue)
{
    FreeBlock(queue);
}

/*
 * Function: Enqueue
 * -----
 * This function adds a new element to the queue, advancing
 * the tail index.
 */

void Enqueue(queueADT queue, void *obj)
{
    if ((queue->tail + 1) % QueueArraySize == queue->head) {
        Error("Enqueue called on a full queue");
    }
    queue->array[queue->tail++] = obj;
    queue->tail %= QueueArraySize;
}

/*
 * Function: Dequeue
 * -----
 * This function removes and returns the data value at the head of
 * the queue.
 */

void *Dequeue(queueADT queue)
{
    void *result;

    if (queue->head == queue->tail) {
        Error("Dequeue of empty queue");
    }
    result = queue->array[queue->head++];
    queue->head %= QueueArraySize;
    return (result);
}

/*
 * Function: QueueLength
 * -----
 * This function returns the number of elements in the queue.
 */

int QueueLength(queueADT queue)
{
    return ((QueueArraySize + queue->tail - queue->head)
            % QueueArraySize);
}
```

8.

```

/*
 * File: queue2.c
 * -----
 * This file implements the queue.h abstraction using a
 * ring buffer that expands dynamically when the queue
 * fills.
 */

#include <stdio.h>

#include "genlib.h"
#include "queue.h"

/*
 * Constants:
 * -----
 * InitialQueueSize -- Initial size reserved for the queue
 */

#define InitialQueueSize 5

/*
 * Type: queueCDT
 * -----
 * This type provides the concrete counterpart to the queueADT.
 * This implementation uses a ring buffer to implement the
 * queue. The next item to be dequeued is found at the array
 * element indexed by head. The tail index indicates the next
 * free position. When head and tail are equal, the queue is
 * empty. The head and tail indices each move from the end of
 * the array back to the beginning, giving rise to the name
 * "ring buffer."
 */

struct queueCDT {
    void **array;
    int head;
    int tail;
    int size;
};

/* Private function prototypes */

static void ExpandQueue(queueADT queue);

/* Exported entries */

/*
 * Function: NewQueue
 * -----
 * This function allocates and initializes the storage for a new queue.
 */

queueADT NewQueue(void)
{
    queueADT queue;

    queue = New(queueADT);
    queue->head = queue->tail = 0;
    queue->size = InitialQueueSize;
    queue->array = NewArray(InitialQueueSize, void *);
    return (queue);
}

```

```
/*
 * Function: FreeQueue
 * -----
 * This function simply frees the queue storage.
 */

void FreeQueue(queueADT queue)
{
    FreeBlock(queue);
}

/*
 * Function: Enqueue
 * -----
 * This function adds a new element to the queue, advancing
 * the tail index.
 */

void Enqueue(queueADT queue, void *obj)
{
    if ((queue->tail + 1) % queue->size == queue->head) {
        ExpandQueue(queue);
    }
    queue->array[queue->tail++] = obj;
    queue->tail %= queue->size;
}


/*
 * Function: Dequeue
 * -----
 * This function removes and returns the data value at the head of
 * the queue.
 */

void *Dequeue(queueADT queue)
{
    void *result;

    if (queue->head == queue->tail) {
        Error("Dequeue of empty queue");
    }
    result = queue->array[queue->head++];
    queue->head %= queue->size;
    return (result);
}

/*
 * Function: QueueLength
 * -----
 * This function returns the number of elements in the queue.
 */

int QueueLength(queueADT queue)
{
    return ((queue->size + queue->tail - queue->head)
            % queue->size);
}
```



```
/*
 * Function: ExpandQueue
 * -----
 * This function doubles the number of elements in the queue
 * and is called automatically when the queue fills.
 */

static void ExpandQueue(queueADT queue)
{
    int newsize, i, n;
    void **newarray;

    newsize = queue->size * 2;
    newarray = NewArray(newsize, void *);
    n = QueueLength(queue);
    for (i = 0; i < n; i++) {
        newarray[i] = queue->array[(queue->head + i) % queue->size];
    }
    FreeBlock(queue->array);
    queue->array = newarray;
    queue->size = newsize;
    queue->head = 0;
    queue->tail = n;
}
```

9.

```

/*
 * File: dict.h
 * -----
 * This interface exports functions for defining and looking
 * up words in a dictionary. This package allows the client
 * to define multiple dictionaries, each of which is represented
 * as an abstract data type.
 */

#ifndef _dict_h
#define _dict_h

#include "genlib.h"

/*
 * Type: dictionaryADT
 * -----
 * This abstract type represents a dictionary and is capable
 * of storing word/definition pairs.
 */

typedef struct dictionaryCDT *dictionaryADT;

/*
 * Function: NewDictionary
 * Usage: dict = NewDictionary();
 * -----
 * This function creates and returns an empty dictionary.
 */

dictionaryADT NewDictionary(void);

/*
 * Function: FreeDictionary
 * Usage: FreeDictionary(dict);
 * -----
 * This function frees the storage associated with dict.
 */

void FreeDictionary(dictionaryADT dict);

/*
 * Function: Define
 * Usage: Define(dict, word, definition);
 * -----
 * This function defines word to have the specified definition. Any
 * previous definition for word is lost. If defining this word would
 * exceed the capacity of the dictionary, an error is generated.
 */

void Define(dictionaryADT dict, string word, string definition);

/*
 * Function: Lookup
 * Usage: str = Lookup(dict, word);
 * -----
 * This function looks up the word in the dictionary and returns its
 * definition. If the word has not been defined, Lookup returns NULL.
 */

string Lookup(dictionaryADT dict, string word);

#endif

```

```
/*
 * File: dict.c
 * -----
 * This file implements the extended dict.h interface.
 */

#include <stdio.h>
#include "genlib.h"
#include "strlib.h"
#include "dict.h"

/*
 * Constant: InitialDictionarySize
 * -----
 * This constant specifies the initial size of the dictionary array.
 * To a large extent, the setting of this parameter is invisible to
 * the client because the dictionary expands if it runs out of space.
 */

#define InitialDictionarySize 50

/*
 * Type: dictEntryT
 * -----
 * This type holds a single entry in the dictionary. Each
 * entry consists of a key and value pair.
 */

typedef struct {
    string key, value;
} dictEntryT;

/*
 * Type: dictionaryCDT
 * -----
 * This structure type is the concrete representation of the
 * dictionaryADT.
 */

struct dictionaryCDT {
    dictEntryT *array;
    int dictSize;
    int nEntries;
};

/* Private function prototypes */

static int FindEntry(dictionaryADT dict, string word);
static void ExpandDictionary(dictionaryADT dict);

/* Exported entries */

dictionaryADT NewDictionary(void)
{
    dictionaryADT dict;

    dict = New(dictionaryADT);
    dict->array = NewArray(InitialDictionarySize, dictEntryT);
    dict->dictSize = InitialDictionarySize;
    dict->nEntries = 0;
    return (dict);
}
```



```
void FreeDictionary(dictionaryADT dict)
{
    FreeBlock(dict->array);
    FreeBlock(dict);
}

void Define(dictionaryADT dict, string word, string definition)
{
    int entry;

    entry = FindEntry(dict, word);
    if (entry == -1) {
        if (dict->nEntries == dict->dictSize) ExpandDictionary(dict);
        entry = dict->nEntries++;
        dict->array[entry].key = CopyString(word);
    }
    dict->array[entry].value = CopyString(definition);
}

string Lookup(dictionaryADT dict, string word)
{
    int entry;


    entry = FindEntry(dict, word);
    if (entry == -1) return (NULL);
    return (dict->array[entry].value);
}

/* Private functions */

/*
 * Function: FindEntry
 * Usage: entry = FindEntry(dict, word);
 * -----
 * This function searches the dictionary dict to find the specified
 * word. If the word exists, FindEntry returns the index in the
 * dictionary array at which it appears. If the word is not in
 * the dictionary, FindEntry returns -1.
 */

static int FindEntry(dictionaryADT dict, string word)
{
    int i;

    for (i = 0; i < dict->nEntries; i++) {
        if (StringEqual(word, dict->array[i].key)) return (i);
    }
    return (-1);
}
```



```
/*
 * Function: ExpandDictionary
 * Usage: ExpandDictionary(dict);
 * -----
 * This function doubles the size of the internal array used to
 * store the dictionary entries.
 */

static void ExpandDictionary(dictionaryADT dict)
{
    dictEntryT *oldWords, *newWords;
    int i, oldSize, newSize;

    oldSize = dict->dictSize;
    newSize = 2 * oldSize;
    oldWords = dict->array;
    newWords = NewArray(newSize, dictEntryT);
    for (i = 0; i < oldSize; i++) {
        newWords[i] = oldWords[i];
    }
    FreeBlock(oldWords);
    dict->array = newWords;
    dict->dictSize = newSize;
}
```

10.

```
/*
 * File: hybsort.c
 * -----
 * This file implements the sort.h interface using a hybrid of
 * the merge and selection sort algorithms.
 */

#include <stdio.h>
#include "genlib.h"
#include "sort.h"

/*
 * Constants
 * -----
 * CrossOver -- Number of elements at which the N log N MergeSort
 *              begins to dominate the quadratic SelectionSort.
 */

#define CrossOver 15

/* Private function prototypes */

void SelectionSort(int array[], int n);
static int FindSmallestInteger(int array[], int low, int high);
static void SwapIntegerElements(int array[], int p1, int p2);
void MergeSort(int array[], int n);
static void Merge(int array[], int arr1[], int n1,
                  int arr2[], int n2);

/*
 * Function: SortIntegerArray
 * -----
 * This implementation decides whether to use MergeSort or
 * SelectionSort based on the size of the array.
 */

void SortIntegerArray(int array[], int n)
{
    if (n < CrossOver) {
        SelectionSort(array, n);
    } else {
        MergeSort(array, n);
    }
}

/*
 * The remaining functions are as given in the text, except that the
 * names of the two implementations of SortIntegerArray are renamed
 * to be SelectionSort and MergeSort.
 */
```

## Section 8

### Related Papers

The following papers offer further information about the educational strategies and tools used in *The Art and Science of C*:

1. Eric Roberts, "Using C in CS1: Evaluating the Stanford experience," presented at the 24th ACM Computer Science Education Conference, February 1993. This paper describes the rationale behind the library-based approach to teaching C and discusses Stanford's early experience using it.
2. Eric Roberts, "A C-based graphics library for CS1," presented at the 26th ACM Computer Science Education Conference, March 1995. The paper focuses on the design of the graphics library and how it achieves its goals of simplicity and portability.
3. Eric Roberts, "Loop exits and structured programming: Reopening the debate," presented at the 26th ACM Computer Science Education Conference, March 1995. This paper defends the position that students are more apt to write correct programs if they are allowed to exit from the interior of a loop in certain constrained situations.
4. Eric Roberts, John Lilly, and Bryan Rollins, "Using undergraduates as teaching assistants in introductory programming courses: An update on the Stanford experience," presented at the 26th ACM Computer Science Education Conference, March 1995. This paper discusses Stanford's use of advanced undergraduates to provide teaching support for the CS1 course.