4.21 Rewrite the program of Fig. 4.2 so that the initialization of the variable **counter** is done in the declaration instead of the **for** structure.

4.22 Modify the program of Fig. 4.7 so that it calculates the average grade for the class.

4.23 Modify the program in Fig. 4.6 so that it uses only integers to calculate the compound interest. (Hint: Treat all monetary amounts as integral numbers of pennies. Then "break" the result into its dollar portion and cents portion by using the division and modulus operations respectively. Insert a period.)

4.24 Assume **i = 1**, **j = 2**, **k = 3** and **m = 2**. What does each of the following statements print?

a) `printf( "%d", i == 1 );`
b) `printf( "%d", j == 3 );`
c) `printf( "%d", i >= 1 && j < 4 );`
d) `printf( "%d", m <= 99 && k < m );`
e) `printf( "%d", j >= i || k == m );`
f) `printf( "%d", k + m < j || 3 - j >= k );`
g) `printf( "%d", !m );`
h) `printf( "%d", !( j - m ) );`
i) `printf( "%d", !( k > m ) );`
j) `printf( "%d", !( j > k ) );`

4.25 Print a table of decimal, binary, octal, and hexadecimal equivalents. If you are not familiar with these number systems, read Appendix E first if you would like to attempt this exercise.

4.26 Calculate the value of π from the infinite series

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \cdots$$

Print a table that shows the value of π approximated by 1 term of this series, by two terms, by three terms, etc. How many terms of this series do you have to use before you first get 3.14? 3.141? 3.1415? 3.14159?

4.27 *(Pythagorean Triples)* A right triangle can have sides that are all integers. The set of three integer values for the sides of a right triangle is called a Pythagorean triple. These three sides must satisfy the relationship that the sum of the squares of two of the sides is equal to the square of the hypotenuse. Find all Pythagorean triples for side1, side2, and the hypotenuse all no larger than 500. Use a triple-nested **for**-loop that simply tries all possibilities. This is an example of "brute force" computing. It is not aesthetically pleasing to many people. But there are many reasons why these techniques are important. First, with computing power increasing at such a phenomenal pace, solutions that would have taken years or even centuries of computer time to produce with the technology of just a few years ago, can now be produced in hours, minutes or even seconds. Recent microprocessor chips can process hundreds of million instructions per second! Second, as you will learn in more advanced computer science courses, there are large numbers of interesting problems for which there is no known algorithmic approach other than sheer brute force. We investigate many kinds of problem-solving methodologies in this book. We will consider many brute force approaches to various interesting problems.

4.28 A company pays its employees as managers (who receive a fixed weekly salary), hourly workers (who receive a fixed hourly wage for up to the first 40 hours they work and "time-and-a-half," i.e., 1.5 times their hourly wage, for overtime hours worked), commission workers (who receive a $250 plus 5.7% of their gross weekly sales), or pieceworkers (who receive a fixed amount of money per item for each of the items they produce—each pieceworker in this company works on only one type of item). Write a program to compute the weekly pay for each employee. You do not know the number of employees in advance. Each type of employee has its own pay code: Managers have

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

5.20 Modify the function created in Exercise 5.19 to form the square out of whatever character is contained in character parameter **fillCharacter**. Thus if **side** is **5** and **fillCharacter** is "#" then this function should print

```
#####
#####
#####
#####
#####
```

5.21 Use techniques similar to those developed in Exercises 5.19 and 5.20 to produce a program that graphs a wide range of shapes.

5.22 Write program segments that accomplish each of the following:

a) Calculate the integer part of the quotient when integer **a** is divided by integer **b**.
b) Calculate the integer remainder when integer **a** is divided by integer **b**.
c) Use the program pieces developed in a) and b) to write a function that inputs an integer between **1** and **32767** and prints it as a series of digits, each pair of which is separated by two spaces. For example, the integer **4562** should be printed as

```
4  5  6  2
```

5.23 Write a function that takes the time as three integer arguments (for hours, minutes, and seconds), and returns the number of seconds since the last time the clock "struck 12." Use this function to calculate the amount of time in seconds between two times, both of which are within one 12-hour cycle of the clock.

5.24 Implement the following integer functions:

a) Function **celsius** returns the Celsius equivalent of a Fahrenheit temperature.
b) Function **fahrenheit** returns the Fahrenheit equivalent of a Celsius temperature.
c) Use these functions to write a program that prints charts showing the Fahrenheit equivalents of all Celsius temperatures from 0 to 100 degrees, and the Celsius equivalents of all Fahrenheit temperatures from 32 to 212 degrees. Print the outputs in a neat tabular format that minimizes the number of lines of output while remaining readable.

5.25 Write a function that returns the smallest of three floating point numbers.

5.26 An integer number is said to be a *perfect number* if its factors, including 1 (but not the number itself), sum to the number. For example, 6 is a perfect number because 6 = 1 + 2 + 3. Write a function **perfect** that determines if parameter **number** is a perfect number. Use this function in a program that determines and prints all the perfect numbers between 1 and 1000. Print the factors of each perfect number to confirm that the number is indeed perfect. Challenge the power of your computer by testing numbers much larger than 1000.

that each succeeding term is the sum of the two
n **fibonacci(n)** that calculates the **n**th
cci number that can be printed on your system.
d of **int** to calculate and return Fibonacci num-
ı excessively high value.

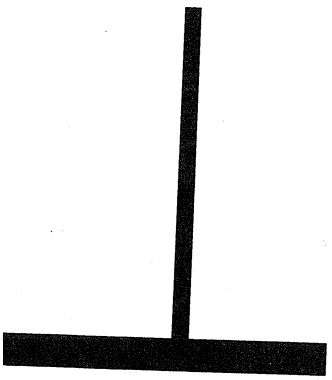
ıter scientist must grapple with certain classic
one of the most famous of these. Legend has it
o move a stack of disks from one peg to another.
and arranged from bottom to top by decreasing
n this peg to a second peg under the constraints
ne may a larger disk be placed above a smaller
the disks. Supposedly the world will end when
ive for us to facilitate their efforts.

nove the disks from peg 1 to peg 3. We wish to
ıce of disk-to-disk peg transfers.

entional methods, we would rapidly find our-
ıstead, if we attack the problem with recursion
lisks can be viewed in terms of moving only *n*

peg 3 as a temporary holding area.
) peg 3.
ng peg 1 as a temporary holding area.
oving $n = 1$ disk, i.e., the base case. This is
ıeed for a temporary holding area.
oroblem. Use a recursive function with four

:aded



four disks.

3. The peg to which this stack of disks is to be moved
4. The peg to be used as a temporary holding area

Your program should print the precise instructions it will take to move the disks from the starting peg to the destination peg. For example, to move a stack of three disks from peg 1 to peg 3, your program should print the following series of moves:

$1 \rightarrow 3$ (This means move one disk from peg 1 to peg 3.)
$1 \rightarrow 2$
$3 \rightarrow 2$
$1 \rightarrow 3$
$2 \rightarrow 1$
$2 \rightarrow 3$
$1 \rightarrow 3$

5.40 Any program that can be implemented recursively can be implemented iteratively, although sometimes with considerably more difficulty and considerably less clarity. Try writing an iterative version of the Towers of Hanoi. If you succeed, compare your iterative version with the recursive version you developed in Exercise 5.39. Investigate issues of performance, clarity, and your ability to demonstrate the correctness of the programs.

5.41 (*Visualizing Recursion*) It is interesting to watch recursion "in action." Modify the factorial function of Fig. 5.14 to print its local variable and recursive call parameter. For each recursive call, display the outputs on a separate line and add a level of indentation. Do your utmost to make the outputs clear, interesting, and meaningful. Your goal here is to design and implement an output format that helps a person understand recursion better. You may want to add such display capabilities to the many other recursion examples and exercises throughout the text.

5.42 The greatest common divisor of integers **x** and **y** is the largest integer that evenly divides both **x** and **y**. Write a recursive function **gcd** that returns the greatest common divisor of **x** and **y**. The **gcd** of **x** and **y** is defined recursively as follows: If **y** is equal to **0**, then **gcd(x, y)** is **x**; otherwise **gcd(x, y)** is **gcd(y, x % y)** where **%** is the modulus operator.

5.43 Can **main** be called recursively? Write a program containing a function **main**. Include **static** local variable **count** initialized to 1. Postincrement and print the value of **count** each time **main** is called. Run your program. What happens?

5.44 Exercises 5.32 through 5.34 developed a computer-assisted instruction program to teach an elementary school student multiplication. This exercise suggests enhancements to that program.

a) Modify the program to allow the user to enter a grade-level capability. A grade level of 1 means to use only single-digit numbers in the problems, a grade level of two means to use numbers as large as two-digits, etc.
b) Modify the program to allow the user to pick the type of arithmetic problems he or she wishes to study. An option of 1 means addition problems only, 2 means subtraction problems only, 3 means multiplication problems only, 4 means division problems only, and 5 means to randomly intermix problems of all these types.

5.45 Write function **distance** that calculates the distance between two points *(x1, y1)* and *(x2, y2)*. All numbers and return values should be of type **double**.

5.46 What does the following program do?

```
1  /* ex05_46.c */
2  #include <stdio.h>
3
```

Chapter 6

Command	Meaning
1	Pen up
2	Pen down
3	Turn right
4	Turn left
5,10	Move forward 10 spaces (or a number other than 10)
6	Print the 20-by-20 array
9	End of data (sentinel)

Suppose that the turtle is somewhere near the center of the floor. The following "program" would draw and print a 12-by 12-square:

```
2
5,12
3
5,12
3
5,12
3
5,12
1
6
9
```

As the turtle moves with the pen down, set the appropriate elements of array **floor** to **1**s. When the **6** command (print) is given, wherever there is a 1 in the array, display an asterisk, or some other character you choose. Wherever there is a zero, display a blank. Write a program to implement the turtle graphics capabilities discussed here. Write several turtle graphics programs to draw interesting shapes. Add other commands to increase the power of your turtle graphics language.

6.24 (*Knight's Tour*) One of the more interesting puzzlers for chess buffs is the Knight's Tour problem, originally proposed by the mathematician Euler. The question is this: Can the chess piece called the knight move around an empty chessboard and touch each of the 64 squares once and only once? We study this intriguing problem in depth here.

The knight makes L-shaped moves (over two in one direction and then over one in a perpendicular direction). Thus, from a square in the middle of an empty chessboard, the knight can make eight different moves (numbered 0 through 7) as shown in Fig. 6.24.

a) Draw an 8-by-8 chessboard on a sheet of paper and attempt a Knight's Tour by hand. Put a **1** in the first square you move to, a **2** in the second square, a **3** in the third, etc. Before starting the tour, estimate how far you think you will get, remembering that a full tour consists of 64 moves. How far did you get? Were you close to the estimate?

b) Now let us develop a program that will move the knight around a chessboard. The board itself is represented by an 8-by-8 double-subscripted array **board**. Each of the squares is initialized to zero. We describe each of the eight possible moves in terms of both their horizontal and vertical components. For example, a move of type 0 as shown in Fig. 6.24 consists of moving two squares horizontally to the right and one square vertically upward. Move 2 consists of moving one square horizontally to the left and two squares vertically upward. Horizontal moves to the left and vertical moves upward are indicated with negative numbers. The eight moves may be described by two single-subscripted arrays, **horizontal** and **vertical**, as follows:

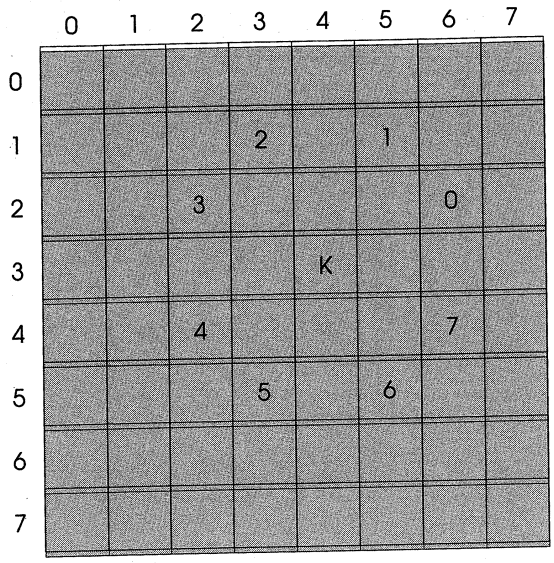


Fig. 6.24 The eight possible moves of the knight.

```
horizontal[ 0 ] = 2
horizontal[ 1 ] = 1
horizontal[ 2 ] = -1
horizontal[ 3 ] = -2
horizontal[ 4 ] = -2
horizontal[ 5 ] = -1
horizontal[ 6 ] = 1
horizontal[ 7 ] = 2

vertical[ 0 ] = -1
vertical[ 1 ] = -2
vertical[ 2 ] = -2
vertical[ 3 ] = -1
vertical[ 4 ] = 1
vertical[ 5 ] = 2
vertical[ 6 ] = 2
vertical[ 7 ] = 1
```

Let the variables **currentRow** and **currentColumn** indicate the row and column of the knight's current position on the board. To make a move of type **moveNumber**, where **moveNumber** is between 0 and 7, your program uses the statements

```
currentRow += vertical[ moveNumber ];
currentColumn += horizontal[ moveNumber ];
```

Keep a counter that varies from **1** to **64**. Record the latest count in each square the knight moves to. Remember to test each potential move to see if the knight has already visited that square. And, of course, test every potential move to make sure that the knight does not land off the chessboard. Now write a program to move the knight around the chessboard. Run the program. How many moves did the knight make?

Chapter 6

c) After attempting to write and run a Knight's Tour program, you have probably developed some valuable insights. We will use these to develop a *heuristic* (or strategy) for moving the knight. Heuristics do not guarantee success, but a carefully developed heuristic greatly improves the chance of success. You may have observed that the outer squares are in some sense more troublesome than the squares nearer the center of the board. In fact, the most troublesome, or inaccessible, squares are the four corners.

Intuition may suggest that you should attempt to move the knight to the most troublesome squares first and leave open those that are easiest to get to so that when the board gets congested near the end of the tour there will be a greater chance of success.

We may develop an "accessibility heuristic" by classifying each of the squares according to how accessible they are and always moving the knight to the square (within the knight's L-shaped moves, of course) that is most inaccessible. We label a double-subscripted array **`accessibility`** with numbers indicating from how many squares each particular square is accessible. On a blank chessboard, the center squares are therefore rated as **8**s, the corner squares are rated as **2**s, and the other squares have accessibility numbers of **3**, **4**, or **6** as follows:

```
2 3 4 4 4 4 3 2
3 4 6 6 6 6 4 3
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
3 4 6 6 6 6 4 3
2 3 4 4 4 4 3 2
```

Now write a version of the Knight's Tour program using the accessibility heuristic. At any time, the knight should move to the square with the lowest accessibility number. In case of a tie, the knight may move to any of the tied squares. Therefore, the tour may begin in any of the four corners. (*Note:* As the knight moves around the chessboard, your program should reduce the accessibility numbers as more and more squares become occupied. In this way, at any given time during the tour, each available square's accessibility number will remain equal to precisely the number of squares from which that square may be reached.) Run this version of your program. Did you get a full tour? Now modify the program to run 64 tours, one from each square of the chessboard. How many full tours did you get?

- Initial Table
- making a move
- update of accessibility table
- case of a tie

d) Write a version of the Knight's Tour program which, when encountering a tie between two or more squares, decides what square to choose by looking ahead to those squares reachable from the "tied" squares. Your program should move to the square for which the next move would arrive at a square with the lowest accessibility number.

6.25 (*Knight's Tour: Brute Force Approaches*) In Exercise 6.24 we developed a solution to the Knight's Tour problem. The approach used, called the "accessibility heuristic," generates many solutions and executes efficiently.

As computers continue increasing in power, we will be able to solve many problems with sheer computer power and relatively unsophisticated algorithms. Let us call this approach "brute force" problem solving.

a) Use random number generation to enable the knight to walk around the chess board (in its legitimate L-shaped moves, of course) at random. Your program should run one tour and print the final chessboard. How far did the knight get?

b) Most likely, the preceding program produced a relatively short tour. Now modify your program to attempt 1000 tours. Use a single-subscripted array to keep track of the number of tours of each length. When your program finishes attempting the 1000 tours, it should print this information in neat tabular format. What was the best result?

c) Most likely, the preceding program gave you some "respectable" tours but no full tours. Now "pull all the stops out" and simply let your program run until it produces a full tour. (*Caution:* This version of the program could run for hours on a powerful computer.) Once again, keep a table of the number of tours of each length and print this table when the first full tour is found. How many tours did your program attempt before producing a full tour? How much time did it take?

d) Compare the brute force version of the Knight's Tour with the accessibility heuristic version. Which required a more careful study of the problem? Which algorithm was more difficult to develop? Which required more computer power? Could we be certain (in advance) of obtaining a full tour with the accessibility heuristic approach? Could we be certain (in advance) of obtaining a full tour with the brute force approach? Argue the pros and cons of brute force problem solving in general.

6.26 (*Eight Queens*) Another puzzler for chess buffs is the Eight Queens problem. Simply stated: Is it possible to place eight queens on an empty chessboard so that no queen is "attacking" any other; that is, so that no two queens are in the same row, the same column, or along the same diagonal? Use the kind of thinking developed in Exercise 6.24 to formulate a heuristic for solving the Eight Queens problem. Run your program. (*Hint:* It is possible to assign a numeric value to each square of the chessboard indicating how many squares of an empty chessboard are "eliminated" once a queen is placed in that square. For example, each of the four corners would be assigned the value 22, as in Fig. 6.25.)

Once these "elimination numbers" are placed in all 64 squares, an appropriate heuristic might be: Place the next queen in the square with the smallest elimination number. Why is this strategy intuitively appealing?

6.27 (*Eight Queens: Brute Force Approaches*) In this problem you will develop several brute force approaches to solving the Eight Queens problem introduced in Exercise 6.26.

a) Solve the Eight Queens problem, using the random brute force technique developed in Problem 6.25.

b) Use an exhaustive technique (i.e., try all possible combinations of eight queens on the chessboard).

c) Why do you suppose the exhaustive brute force approach may not be appropriate for solving the Knight's Tour problem?

d) Compare and contrast the random brute force and exhaustive brute force approaches in general.

6.28 (*Duplicate elimination*) In Chapter 12, we explore the high-speed binary search tree data structure. One feature of a binary search tree is that duplicate values are discarded when insertions are made into the tree. This is referred to as duplicate elimination. Write a program that produces 20 random numbers between 1 and 20. The program should store all nonduplicate values in an array. Use the smallest possible array to accomplish this task.

Fig. 6.25 The 22 squares eliminated by placing a queen in the upper-left corner.