- Objects of standard class template `vector` can be compared directly with the equality (`==`) and inequality (`!=`) operators. The assignment (`=`) operator can also be used with `vector` objects.
- A nonmodifiable *lvalue* (a `const` reference) is an expression that identifies an object in memory (such as an element in a vector), but cannot be used to modify that object. A modifiable *lvalue* (a non-`const` reference) also identifies an object in memory, but can be used to modify the object.
- An **exception** (p. 321) indicates a problem that occurs while a program executes. The name "exception" suggests that the problem occurs infrequently—if the "rule" is that a statement normally executes correctly, then the problem represents the "exception to the rule."
- **Exception handling** (p. 321) enables you to create **fault-tolerant programs** (p. 321) that can resolve exceptions.
- To handle an exception, place the code that might **throw an exception** (p. 322) in a `try` statement.
- The **`try` block** (p. 322) contains the code that might throw an exception, and the **`catch` block** (p. 322) contains the code that handles the exception if one occurs.
- When a `try` block terminates, any variables declared in the `try` block go out of scope.
- A `catch` block declares a type and an exception parameter. Inside the `catch` block, you can use the parameter's identifier to interact with a caught exception object.
- An exception object's **`what` member function** (p. 322) returns the exception's error message.

Self-Review Exercises

7.1 *(Fill in the Blanks)* Answer each of the following:

a) Lists and tables of values can be stored in _____ or _____.
b) An `array`'s elements are related by the fact that they have the same _____ and _____.
c) The number used to refer to a particular element of an `array` is called its _____.
d) A(n) _____ should be used to declare the size of an `array`, because it eliminates magic numbers.
e) The process of placing the elements of an `array` in order is called _____ the `array`.
f) The process of determining if an `array` contains a particular key value is called _____ the `array`.
g) An `array` that uses two subscripts is referred to as a(n) _____ `array`.

7.2 *(True or False)* State whether the following are *true* or *false*. If the answer is *false*, explain why.

a) A given `array` can store many different types of values.
b) An `array` subscript should normally be of data type `float`.
c) If there are fewer initializers in an initializer list than the number of elements in the `array`, the remaining elements are initialized to the last value in the initializer list.
d) It's an error if an initializer list has more initializers than there are elements in the `array`.

7.3 *(Write C++ Statements)* Write one or more statements that perform the following tasks for an `array` called `fractions`:

a) Define a constant variable `arraySize` to represent the size of an `array` and initialize it to 10.
b) Declare an `array` with `arraySize` elements of type `double`, and initialize the elements to 0.
c) Name the fourth element of the `array`.
d) Refer to `array` element 4.
e) Assign the value `1.667` to `array` element 9.
f) Assign the value `3.333` to the seventh element of the `array`.

g) Display array elements 6 and 9 with two digits of precision to the right of the decimal point, and show the output that's actually displayed on the screen.
h) Display all the array elements using a counter-controlled for statement. Define the integer variable i as a control variable for the loop. Show the output.
i) Display all the array elements separated by spaces using a range-based for statement.

7.4 *(Two-Dimensional array Questions)* Answer the following questions regarding a two-dimensional array called table:

a) Declare the array to store int values and to have 3 rows and 3 columns. Assume that the constant variable arraySize has been defined to be 3.
b) How many elements does the array contain?
c) Use a counter-controlled for statement to initialize each element of the array to the sum of its subscripts.
d) Write a nested for statement that displays the values of each element of array table in tabular format with 3 rows and 3 columns. Each row and column should be labeled with the row or column number. Assume that the array was initialized with an initializer list containing the values from 1 through 9 in order. Show the output.

7.5 *(Find the Error)* Find and correct the error in each of the following program segments:

a)
```
#include <iostream>;
```
b)
```
arraySize = 10; // arraySize was declared const
```
c) Assume that
```
array<int, 10> b{};
for (size_t i{0}; i <= b.size(); ++i) {
   b[i] = 1;
}
```
d) Assume that a is a two-dimensional array of int values with two rows and two columns:
```
a[1, 1] = 5;
```

Answers to Self-Review Exercises

7.1 a) arrays, vectors. b) array name, type. c) subscript or index. d) constant variable. e) sorting. f) searching. g) two-dimensional.

7.2 a) False. An array can store only values of the same type.
b) False. An array subscript should be an integer or an integer expression.
c) False. The remaining elements are initialized to zero.
d) True.

7.3 a)
```
const size_t arraySize{10};
```
b)
```
array<double, arraySize> fractions{0.0};
```
c)
```
fractions[3]
```
d)
```
fractions[4]
```
e)
```
fractions[9] = 1.667;
```
f)
```
fractions[6] = 3.333;
```
g)
```
cout << fixed << setprecision(2);
cout << fractions[6] << ' ' << fractions[9] << endl;
```
Output: 3.33 1.67

h)
```
for (size_t i{0}; i < fractions.size(); ++i) {
   cout << "fractions[" << i << "] = " << fractions[i] << endl;
}
```
Output:
```
fractions[0] = 0.0
fractions[1] = 0.0
fractions[2] = 0.0
```

```
fractions[3] = 0.0
fractions[4] = 0.0
fractions[5] = 0.0
fractions[6] = 3.333
fractions[7] = 0.0
fractions[8] = 0.0
fractions[9] = 1.667
```

i)
```
for (double element : fractions)
   cout << element << ' ';
```

7.4 a)
```
array<array<int, arraySize>, arraySize> table;
```

b) Nine.

c)
```
for (size_t row{0}; row < table.size(); ++row) {
   for (size_t column{0}; column < table[row].size(); ++column) {
      table[row][column] = row + column;
   }
}
```

d)
```
cout << "      [0]  [1]  [2]" << endl;
for (size_t i{0}; i < arraySize; ++i) {
   cout << '[' << i << "] ";
   for (size_t j{0}; j < arraySize; ++j) {
      cout << setw(3) << table[i][j] << "  ";
   }
   cout << endl;
}
```

Output:

```
     [0]  [1]  [2]
[0]   1    2    3
[1]   4    5    6
[2]   7    8    9
```

7.5 a) *Error:* Semicolon at end of `#include` preprocessing directive.
Correction: Eliminate semicolon.

b) *Error:* Assigning a value to a constant variable using an assignment statement.
Correction: Initialize the constant variable in a `const size_t arraySize` declaration.

c) *Error:* Referencing an `array` element outside the bounds of the `array` (`b[10]`).
Correction: Change the loop-continuation condition to use < rather than <=.

d) *Error:* `array` subscripting done incorrectly.
Correction: Change the statement to `a[1][1] = 5;`

Exercises

7.6 ***(Fill in the Blanks)*** Fill in the blanks in each of the following:

a) The names of the four elements of `array p` are _____, _____, _____ and _____.

b) Naming an `array`, stating its type and specifying the number of elements in the `array` is called _____ the `array`.

c) When accessing an `array` element, by convention, the first subscript in a two-dimensional `array` identifies an element's _____ and the second subscript identifies an element's _____.

d) An *m*-by-*n* `array` contains _____ rows, _____ columns and _____ elements.

e) The name of the element in row 3 and column 5 of `array d` is _____.

7.7 *(True or False)* Determine whether each of the following is *true* or *false*. If *false*, explain why.

a) To refer to a particular location or element within an `array`, we specify the name of the `array` and the value of the particular element.

b) An `array` definition reserves space for an `array`.

c) To reserve 100 locations for integer `array p`, you write

```
p[100];
```

d) A `for` statement must be used to initialize the elements of a 15-element `array` to zero.

e) Nested `for` statements must be used to total the elements of a two-dimensional `array`.

7.8 *(Write C++ Statements)* Write C++ statements to accomplish each of the following:

a) Display the value of element 6 of character `array alphabet`.

b) Input a value into element 4 of one-dimensional floating-point `array grades`.

c) Initialize each of the 5 elements of one-dimensional integer `array values` to 8.

d) Total and display the elements of floating-point `array temperatures` of 100 elements.

e) Copy `array a` into the first portion of `array b`. Assume that both `arrays` contain `doubles` and that `arrays a` and `b` have 11 and 34 elements, respectively.

f) Determine and display the smallest and largest values contained in 99-element floating-point `array w`.

7.9 *(Two-Dimensional array Questions)* Consider a 2-by-3 integer `array t`.

a) Write a declaration for `t`.

b) How many rows does `t` have?

c) How many columns does `t` have?

d) How many elements does `t` have?

e) Write the names of all the elements in row 1 of `t`.

f) Write the names of all the elements in column 2 of `t`.

g) Write a statement that sets the element of `t` in the first row and second column to zero.

h) Write a series of statements that initialize each element of `t` to zero. Do not use a loop.

i) Write a nested counter-controlled `for` statement that initializes each element of `t` to zero.

j) Write a nested range-based `for` statement that initializes each element of `t` to zero.

k) Write a statement that inputs the values for the elements of `t` from the keyboard.

l) Write a series of statements that determine and display the smallest value in `array t`.

m) Write a statement that displays the elements in row 0 of `t`.

n) Write a statement that totals the elements in column 2 of `t`.

o) Write a series of statements that prints the `array t` in neat, tabular format. List the column subscripts as headings across the top and list the row subscripts at the left of each row.

7.10 *(Salesperson Salary Ranges)* Use a one-dimensional `array` to solve the following problem. A company pays its salespeople on a commission basis. The salespeople each receive $200 per week plus 9 percent of their gross sales for that week. For example, a salesperson who grosses $5000 in sales in a week receives $200 plus 9 percent of $5000, or a total of $650. Write a program (using an `array` of counters) that determines how many of the salespeople earned salaries in each of the following ranges (assume that each salesperson's salary is truncated to an integer amount):

a) $200–299

b) $300–399

c) $400–499

d) $500–599

e) $600–699

f) $700–799

g) $800–899

h) $900–999

i) $1000 and over

7.11 *(One-Dimensional array Questions)* Write statements that perform the following one-dimensional array operations:

a) Initialize the 10 elements of integer array counts to zero.
b) Add 1 to each of the 15 elements of integer array bonus.
c) Read 12 values for the array of doubles named monthlyTemperatures from the keyboard.
d) Print the 5 values of integer array bestScores in column format.

7.12 *(Find the Errors)* Find the error(s) in each of the following statements:

a) Assume that a is an array of three ints.

```
cout << a[1] << " " << a[2] << " " << a[3] << endl;
```

b) `array<double, 3> f{1.1, 10.01, 100.001, 1000.0001};`
c) Assume that d is an array of doubles with two rows and 10 columns.

```
d[1, 9] = 2.345;
```

7.13 *(Duplicate Elimination with array)* Use a one-dimensional array to solve the following problem. Read in 20 numbers, each of which is between 10 and 100, inclusive. As each number is read, validate it and store it in the array only if it isn't a duplicate of a number already read. After reading all the values, display only the unique values that the user entered. Provide for the "worst case" in which all 20 numbers are different. Use the smallest possible array to solve this problem.

7.14 *(Duplicate Elimination with vector)* Reimplement Exercise 7.13 using a vector. Begin with an empty vector and use its push_back function to add each unique value to the vector.

7.15 *(Two-Dimensional array Initialization)* Label the elements of a 3-by-5 two-dimensional array sales to indicate the order in which they're set to zero by the following program segment:

```
for (size_t row{0}; row < sales.size(); ++row) {
   for (size_t column{0}; column < sales[row].size(); ++column) {
      sales[row][column] = 0;
   }
}
```

7.16 *(Dice Rolling)* Write a program that simulates the rolling of two dice. The sum of the two values should then be calculated. [*Note:* Each die can show an integer value from 1 to 6, so the sum of the two values will vary from 2 to 12, with 7 being the most frequent sum and 2 and 12 being the least frequent sums.] Figure 7.22 shows the 36 possible combinations of the two dice. Your program should roll the two dice 36,000,000 times. Use a one-dimensional array to tally the numbers of times each possible sum appears. Print the results in a tabular format. Also, determine if the totals are reasonable (i.e., there are six ways to roll a 7, so approximately one-sixth of all the rolls should be 7).

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Fig. 7.22 | The 36 possible outcomes of rolling two dice.

7.17 (W

```
1  // Ex
2  // Wha
3  #inclu
4  #inclu
5  using
6
7  const
8  int wh
9
10 int ma
11    arr
12
13    int
14
15    cou
16 }
17
18 // Wha
19 int wh
20    if
21
22    }
23    els
24
25    }
26 }
```

Fig. 7.23 |

7.18 (C
The progra
a)
b)
c)

d)
e)

7.19 (C
Fig. 7.21 t

7.20 (W

```
1  // Ex.
2  // Wha
3  #inclu
4  #inclu
5  using
6
7  const
8  void s
9
```

Fig. 7.24 |

7.17 *(What Does This Code Do?)* What does the following program do?

```
// Ex. 7.17: Ex07_17.cpp
// What does this program do?
#include <iostream>
#include <array>
using namespace std;

const size_t arraySize{10};
int whatIsThis(const array<int, arraySize>&, size_t); // prototype

int main() {
   array<int, arraySize> a{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

   int result{whatIsThis(a, arraySize)};

   cout << "Result is " << result << endl;
}

// What does this function do?
int whatIsThis(const array<int, arraySize>& b, size_t size) {
   if (size == 1) { // base case
      return b[0];
   }
   else { // recursive step
      return b[size - 1] + whatIsThis(b, size - 1);
   }
}
```

Fig. 7.23 | What does this program do?

7.18 *(Craps Game Modification)* Modify the program of Fig. 6.9 to play 1000 games of craps. The program should keep track of the statistics and answer the following questions:

a) How many games are won on the 1st roll, 2nd roll, ..., 20th roll, and after the 20th roll?
b) How many games are lost on the 1st roll, 2nd roll, ..., 20th roll, and after the 20th roll?
c) What are the chances of winning at craps? [*Note:* You should discover that craps is one of the fairest casino games. What do you suppose this means?]
d) What's the average length of a game of craps?
e) Do the chances of winning improve with the length of the game?

7.19 (*Converting* `vector` *Example of Section 7.10 to* `array`) Convert the `vector` example of Fig. 7.21 to use `array`s. Eliminate any `vector`-only features.

7.20 *(What Does This Code Do?)* What does the following program do?

```
// Ex. 7.20: Ex07_20.cpp
// What does this program do?
#include <iostream>
#include <array>
using namespace std;

const size_t arraySize{10};
void someFunction(const array<int, arraySize>&, size_t); // prototype

```

Fig. 7.24 | What does this program do? (Part 1 of 2.)

```
10  int main() {
11     array<int, arraySize> a{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
12
13     cout << "The values in the array are:" << endl;
14     someFunction(a, 0);
15     cout << endl;
16  }
17
18  // What does this function do?
19  void someFunction(const array<int, arraySize>& b, size_t current) {
20     if (current < b.size()) {
21        someFunction(b, current + 1);
22        cout << b[current] << "  ";
23     }
24  }
```

Fig. 7.24 | What does this program do? (Part 2 of 2.)

7.21 *(Sales Summary)* Use a two-dimensional array to solve the following problem. A company has four salespeople (1 to 4) who sell five different products (1 to 5). Once a day, each salesperson passes in a slip for each different type of product sold. Each slip contains the following:

a) The salesperson number
b) The product number
c) The total dollar value of that product sold that day

Thus, each salesperson passes in between 0 and 5 sales slips per day. Assume that the information from all of the slips for last month is available. Write a program that will read all this information for last month's sales (one salesperson's data at a time) and summarize the total sales by salesperson by product. All totals should be stored in the two-dimensional array `sales`. After processing all the information for last month, print the results in tabular format with each of the columns representing a particular salesperson and each of the rows representing a particular product. Cross total each row to get the total sales of each product for last month; cross total each column to get the total sales by salesperson for last month. Your tabular printout should include these cross totals to the right of the totaled rows and to the bottom of the totaled columns.

7.22 (*Knight's Tour*) One of the more interesting puzzlers for chess buffs is the Knight's Tour problem. 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 in this exercise.

The knight makes L-shaped moves (over two in one direction 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. 7.25.

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'll get, remembering that a full tour consists of 64 moves. How far did you get? Was this close to your estimate?
b) Now let's develop a program that will move the knight around a chessboard. The board is represented by an 8-by-8 two-dimensional 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. 7.25, 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

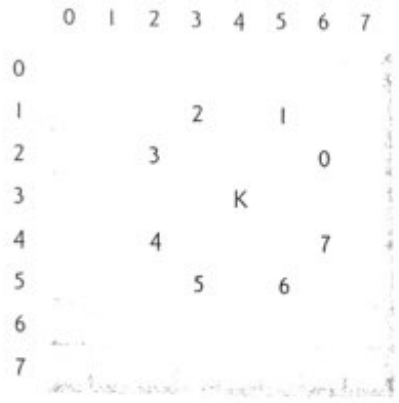

Fig. 7.25 | The eight possible moves of the knight.

vertically upward. Horizontal moves to the left and vertical moves upward are indicated with negative numbers. The eight moves may be described by two one-dimensional arrays, `horizontal` and `vertical`, as follows:

```
horizontal[ ] =          vertical[ ] =
horizontal[ ] =          vertical[ ] =
horizontal[ ] =          vertical[ ] =
horizontal[ ] =          vertical[ ] =
horizontal[ ] =          vertical[ ] =
horizontal[ ] =          vertical[ ] =
horizontal[ ] =          vertical[ ] =
horizontal[ ] =          vertical[ ] =
```

Let the variables `currentRow` and `currentColumn` indicate the row and column of the knight's current position. 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?

c) After attempting to write and run a Knight's Tour program, you've probably developed some valuable insights. We'll 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 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 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 square according to how accessible it's then always moving the knight to the square (within the knight's L-shaped moves, of course) that's least accessible. We label a two-dimensional `array accessibility` with numbers indicating from how many squares each particular square is accessible. On a blank chessboard, each center square is rated as 8, each corner square is rated as 2 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 starting from each square of the chessboard. How many full tours did you get?

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.

7.23 (*Knight's Tour: Brute Force Approaches*) In Exercise 7.22, 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'll be able to solve more problems with sheer computer power and relatively unsophisticated algorithms. This is the "brute force" approach to problem solving.

a) Use random number generation to enable the knight to walk around the chessboard (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 one-dimensional `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.

7.24 (*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, i.e., no two queens are in the same row, the same column, or along the same diagonal? Use the thinking developed in Exercise 7.22 to formulate a heuristic for solving the Eight Queens problem. Run

your program. [*Hint:* It's possible to assign a value to each square of the chessboard indicating how many squares of an empty chessboard are "eliminated" if a queen is placed in that square. Each of the corners would be assigned the value 22, as in Fig. 7.26. 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?]

Fig. 7.26 | The 22 squares eliminated by placing a queen in the upper-left corner.

7.25 (*Eight Queens: Brute Force Approaches*) In this exercise, you'll develop several brute-force approaches to solving the Eight Queens problem introduced in Exercise 7.24.

a) Solve the Eight Queens exercise, using the random brute force technique developed in Exercise 7.23.
b) Use an exhaustive technique, i.e., try all possible combinations of eight queens.
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 and exhaustive brute force approaches in general.

7.26 (*Knight's Tour: Closed-Tour Test*) In the Knight's Tour, a full tour occurs when the knight makes 64 moves, touching each square of the board once and only once. A closed tour occurs when the 64th move is one move away from the location in which the knight started the tour. Modify the Knight's Tour program you wrote in Exercise 7.22 to test for a closed tour if a full tour has occurred.

7.27 (*The Sieve of Eratosthenes*) A prime integer is any integer that's evenly divisible only by itself and 1. The Sieve of Eratosthenes is a method of finding prime numbers. It operates as follows:

a) Create an `array` with all elements initialized to 1 (true). `array` elements with prime subscripts will remain 1. All other `array` elements will eventually be set to zero. You'll ignore elements 0 and 1 in this exercise.
b) Starting with `array` subscript 2, every time an `array` element is found whose value is 1, loop through the remainder of the `array` and set to zero every element whose subscript is a multiple of the subscript for the element with value 1. For `array` subscript 2, all elements beyond 2 in the `array` that are multiples of 2 will be set to zero (subscripts 4, 6, 8, 10, etc.); for `array` subscript 3, all elements beyond 3 in the `array` that are multiples of 3 will be set to zero (subscripts 6, 9, 12, 15, etc.); and so on.

When this process is complete, the `array` elements that are still set to one indicate that the subscript is a prime number. These can then be printed. Write a program that uses an `array` of 1000 elements to determine and print the prime numbers between 2 and 999.

Recursion Exercises

7.28 (*Palindromes*) A palindrome is a string that's spelled the same way forward and backward. Examples of palindromes include "radar" and "able was i ere i saw elba." Write a recursive function

testPalindrome that returns true if a string is a palindrome, and false otherwise. Note that like an array, the square brackets ([]) operator can be used to iterate through the characters in a string.

7.29 (*Eight Queens*) Modify the Eight Queens program you created in Exercise 7.24 to solve the problem recursively.

7.30 (*Print an array*) Write a recursive function printArray that takes an array, a starting subscript and an ending subscript as arguments, returns nothing and prints the array. The function should stop processing and return when the starting subscript equals the ending subscript.

7.31 (*Print a String Backward*) Write a recursive function stringReverse that takes a string and a starting subscript as arguments, prints the string backward and returns nothing. The function should stop processing and return when the end of the string is encountered. Note that like an array the square brackets ([]) operator can be used to iterate through the characters in a string.

7.32 (*Find the Minimum Value in an array*) Write a recursive function recursiveMinimum that takes an integer array, a starting subscript and an ending subscript as arguments, and returns the smallest element of the array. The function should stop processing and return when the starting subscript equals the ending subscript.

7.33 (*Maze Traversal*) The grid of hashes (#) and dots (.) in Fig. 7.27 is a two-dimensional built-in array representation of a maze. In the two-dimensional built-in array, the hashes represent the walls of the maze and the dots represent squares in the possible paths through the maze. Moves can be made only to a location in the built-in array that contains a dot.

There is a simple algorithm for walking through a maze that guarantees finding the exit (assuming that there is an exit). If there is not an exit, you'll arrive at the starting location again. Place your right hand on the wall to your right and begin walking forward. Never remove your hand from the wall. If the maze turns to the right, you follow the wall to the right. As long as you do not remove your hand from the wall, eventually you'll arrive at the exit of the maze. There may be a shorter path than the one you've taken, but you are guaranteed to get out of the maze if you follow the algorithm.

```
# # # # # # # # # # # #
# . . . # . . . . . . #
. . # . # . # # # # . #
# # # . # . . . . # . #
# . . . . # # # . # . .
# # # # . # . # . # . #
# . . # . # . # . # . #
# # . # . # . # . # . #
# . . . . . . . # . #
# # # # # # . # # # . #
# . . . . . # . . . #
# # # # # # # # # # # #
```

Fig. 7.27 | Two-dimensional built-in array representation of a maze.

Write recursive function mazeTraverse to walk through the maze. The function should receive arguments that include a 12-by-12 built-in array of chars representing the maze and the starting location of the maze. As mazeTraverse attempts to locate the exit from the maze, it should place the character X in each square in the path. The function should display the maze after each move, so the user can watch as the maze is solved.

7.34 (*Generating Mazes Randomly*) Write a function mazeGenerator that randomly produces a maze. The function should take as arguments a two-dimensional 12-by-12 built-in array of chars and references to the int variables that represent the row and column of the maze's entry point. Try your function mazeTraverse from Exercise 7.33, using several randomly generated mazes.