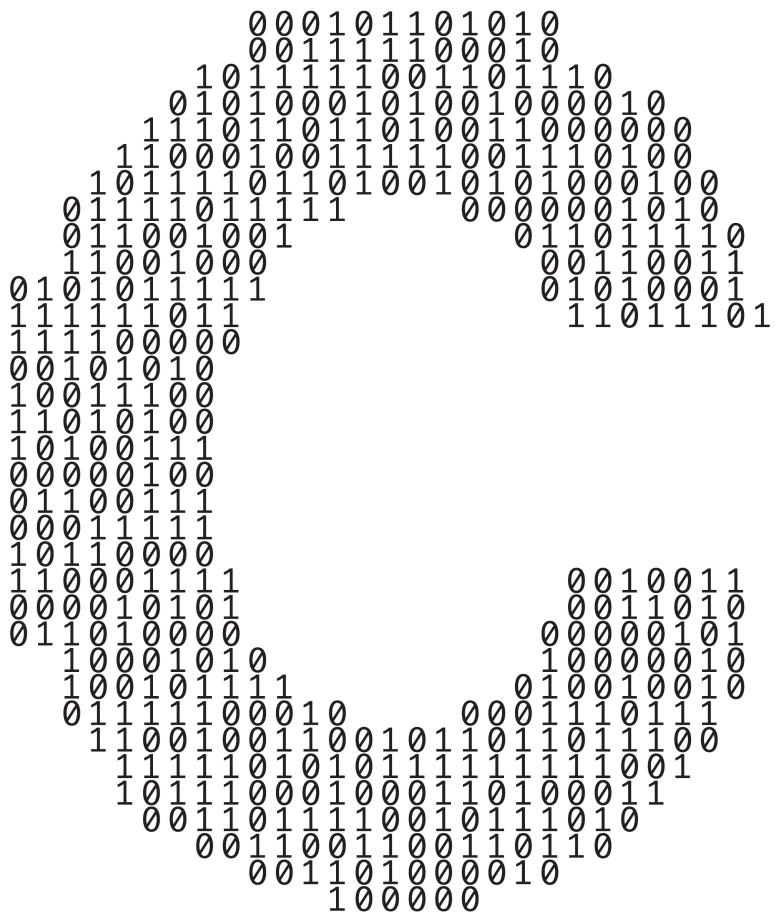
6

Arrays



Objectives

In this chapter, you'll:

- Use the array data structure to represent lists and tables of values.
- Define arrays, initialize arrays and refer to individual elements of arrays.
- Define symbolic constants.
- Pass arrays to functions.
- Use arrays to store, sort and search lists and tables of values.
- Be introduced to data science using basic descriptive statistics, such as mean, median and mode.
- Define and manipulate multidimensional arrays.
- Create variable-length arrays whose size is determined at execution time.
- Understand security issues related to input with `scanf`, output with `printf` and arrays.

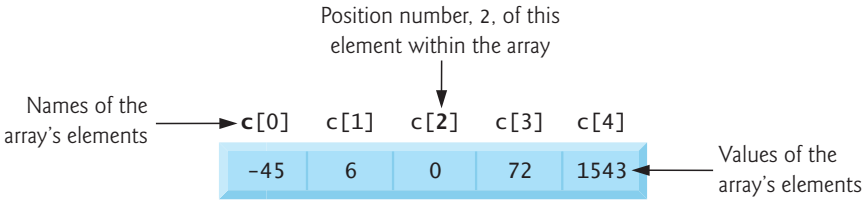
Outline

6.1 Introduction

This chapter introduces data structures. **Arrays** are data structures consisting of related data items of the same type. Chapter 10 discusses C's notion of `struct`—a data structure consisting of related data items of possibly different types. Arrays and `structs` are "static" entities in that they remain the same size throughout their lifetimes.

6.2 Arrays

An array is a group of **elements** of the same type stored contiguously in memory. The following diagram shows an integer array called `c`, containing five elements:



To refer to a particular location or element in the array, we specify the array's name, followed by the element's **position number** in square brackets (`[]`). The first element is located at position number `0` (zero). The position number is called the element's **subscript** (or **index**). A subscript must be a non-negative integer or integer expression.

Let's examine the array in the previous diagram more closely. The array's **name** is `c`. The **value** of `c[0]` is `-45`, `c[2]` is `0` and `c[4]` is `1543`. A subscripted array name is an *lvalue* that can be used on the left side of an assignment. So, the statement:

```
c[2] = 1000;
```

replaces `c[2]`'s current value (`0`) with the value `1000`. To print the sum of the values in array `c`'s first three elements, we'd write:

```
printf("%d", c[0] + c[1] + c[2]);
```

To divide the value of element `3` of array `c` by `2` and assign the result to the variable `x`, write:

```
x = c[3] / 2;
```

The brackets that enclose an array's subscript are an operator with the highest level of precedence. The following table shows the precedence and grouping of the operators introduced to this point in the text:

Operators	Grouping	Type
`[]` `()` *`++(postfix)`* *`--(postfix)`*	left to right	highest
`+` `-` `!` *`++(prefix)`* *`--(prefix)`* *`(type)`*	right to left	unary
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment
`,`	left to right	comma

Self Check

1 ***(Multiple Choice)*** Which of the following statements is *false*?

a) Any array element may be referred to by giving the array's name followed by the element's position number in square brackets (`[]`).

b) Every array's first element has position number `1`.

c) An array name, like other identifiers, can contain only letters, digits and underscores and cannot begin with a digit.

d) The position number in square brackets is called the element's subscript (or index), which must be an integer or an integer expression.

Answer: b) is *false*. Actually, every array's first element has position number `0`.

2 *(Code)* Write a statement that displays the `int` product of the values contained in the first four elements of `int` array `grades`.

Answer: `printf("%d", grades[0] * grades[1] * grades[2] * grades[3]);`

6.3 Defining Arrays

When you define an array, you specify its element type and number of elements so the compiler may reserve the appropriate amount of memory. The following definition reserves five elements for integer array `c`, which has subscripts in the range 0–4.

```
int c[5];
```

The definitions

```
int b[100];
int x[27];
```

reserve 100 elements for integer array `b` and 27 elements for integer array `x`. These arrays have subscripts in the ranges 0–99 and 0–26, respectively.

A `char` array can store a character string. Character strings and their similarity to arrays are discussed in Chapter 8. The relationship between pointers and arrays is discussed in Chapter 7.

Self Check

1 *(Multiple Choice)* Which of the following statements is *false*?

a) When creating an array, you specify an array's element type and number of elements so the compiler may reserve the appropriate amount of memory.

b) The following definition reserves space for the `double` array `temperatures`, which has index numbers in the range `0`–`6`:

```
double temperatures[7];
```

c) The following definitions reserve 50 elements for `float` array `b` and `19` elements for `float` array `x`:

```
float b[50];
float x[19];
```

d) An array of type `string` can store a character string.

Answer: d) is *false*. Actually, an array of type `char` can store a character string. C does not have a `string` type.

6.4 Array Examples

This section presents several examples demonstrating how to define and initialize arrays and how to perform many common array manipulations.

6.4.1 Defining an Array and Using a Loop to Set the Array's Element Values

Like any other local variable, uninitialized array elements contain "garbage" values. Figure 6.1 uses `for` statements to set five-element integer array `n`'s elements to zeros (lines 10–12) and print the array in tabular format (lines 17–19). The first `printf` statement (line 14) displays the column heads for the two columns printed in the subsequent `for` statement.

```
1  // fig06_01.c
2  // Initializing the elements of an array to zeros.
3  #include <stdio.h>
4
5  // function main begins program execution
6  int main(void) {
7     int n[5]; // n is an array of five integers
8
9     // set elements of array n to 0
10    for (size_t i = 0; i < 5; ++i) {
11       n[i] = 0; // set element at location i to 0
12    }
13
14    printf("%s%8s\n", "Element", "Value");
15
16    // output contents of array n in tabular format
17    for (size_t i = 0; i < 5; ++i) {
18       printf("%7zu%8d\n", i, n[i]);
19    }
20 }
```

```
Element   Value
      0       0
      1       0
      2       0
      3       0
      4       0
```

Fig. 6.1 | Initializing the elements of an array to zeros.

The counter-control variable `i`'s type is `size_t` in each `for` statement (lines 10 and 17). The C standard says `size_t` represents an unsigned integral type and is recommended for any variable representing an array's size or subscripts. Type `size_t` is defined in header `<stddef.h>`, which is often included by other headers (such as `<stdio.h>`).[1] The **conversion specification `%zu`** is used to display `size_t` values.

1. If you attempt to compile Fig. 6.1 and receive errors, include `<stddef.h>` in your program.

6.4.2 Initializing an Array in a Definition with an Initializer List

You can initialize an array's elements when defining the array by providing a comma-separated list of **array initializers** in braces, `{}`. Figure 6.2 initializes an integer array with five values (line 7) and prints it in a tabular format.

```
1  // fig06_02.c
2  // Initializing the elements of an array with an initializer list.
3  #include <stdio.h>
4
5  // function main begins program execution
6  int main(void) {
7     int n[5] = {32, 27, 64, 18, 95}; // initialize n with initializer list
8
9     printf("%s%8s\n", "Element", "Value");
10
11    // output contents of array in tabular format
12    for (size_t i = 0; i < 5; ++i) {
13       printf("%7zu%8d\n", i, n[i]);
14    }
15 }
```

```
Element   Value
      0      32
      1      27
      2      64
      3      18
      4      95
```

Fig. 6.2 | Initializing the elements of an array with an initializer list.

If there are fewer initializers than array elements, the remaining elements are initialized to `0`. For example, Fig. 6.1 could have initialized array `n`'s elements to zero as follows:

```
int n[5] = {0}; // initializes entire array to zeros
```

This explicitly initializes `n[0]` to `0` and implicitly initializes the remaining elements to `0`. It's a compilation error if you provide more initializers in an array initializer list than elements in the array. For example, the following array definition produces a compilation error because there are four initializers for only three elements:

ERR

```
int n[3] = {32, 27, 64, 18};
```

The following definition creates a five-element array initialized with the values 1–5:

```
int n[] = {1, 2, 3, 4, 5};
```

When you omit the array size, the compiler calculates the array's number of elements from the number initializers.

6.4.3 Specifying an Array's Size with a Symbolic Constant and Initializing Array Elements with Calculations

Figure 6.3 initializes five-element array `s` with the values `2`, `4`, `6`, `8` and `10`, then prints the array in tabular format. To generate the values, we multiply the loop counter by `2` and add `2`.

```
1  // fig06_03.c
2  // Initializing the elements of array s to the even integers from 2 to 10.
3  #include <stdio.h>
4  #define SIZE 5 // maximum size of array
5
6  // function main begins program execution
7  int main(void) {
8     // symbolic constant SIZE can be used to specify array size
9     int s[SIZE] = {0}; // array s has SIZE elements
10
11    for (size_t j = 0; j < SIZE; ++j) { // set the values
12       s[j] = 2 + 2 * j;
13    }
14
15    printf("%s%8s\n", "Element", "Value");
16
17    // output contents of array s in tabular format
18    for (size_t j = 0; j < SIZE; ++j) {
19       printf("%7zu%8d\n", j, s[j]);
20    }
21 }
```

```
Element   Value
      0       2
      1       4
      2       6
      3       8
      4      10
```

Fig. 6.3 | Initializing the elements of array `s` to the even integers from 2 to 10.

Line 4 uses the **`#define` preprocessor directive**

```
#define SIZE 5
```

to create the **symbolic constant** `SIZE` with the value `5`. A symbolic constant is an identifier that the C preprocessor replaces with **replacement text** before the program is compiled. In this program, the preprocessor replaces all `SIZE` occurrences with `5`.

Using symbolic constants to specify array sizes makes programs easier to read and modify. In Fig. 6.3, for example, we could have the first `for` loop (line 11) fill a 1000-element array simply by changing `SIZE`'s value in the `#define` directive from `5` to `1000`. Without the symbolic constant, we'd have to change the program in lines 9, 11 and 18. As programs get larger, this technique becomes more useful for writing clear, easy-to-read, maintainable programs. A symbolic constant (like `SIZE`) is easier to understand than the numeric value `5`, which could have different meanings throughout the code.

Do not terminate the `#define` preprocessor directives with semicolons. If you do that in line 4, the preprocessor replaces all occurrences of `SIZE` with the text `"5;"`. This may lead to syntax errors at compile time or logic errors at execution time. Remember that the preprocessor is not the C compiler.

ERR

Assigning a value to a symbolic constant in an executable statement is a compilation error—symbolic constants are not variables. By convention, use only uppercase letters for symbolic constant names, so they stand out in a program. This also reminds you that symbolic constants are not variables.

ERR

6.4.4 Summing the Elements of an Array

Figure 6.4 sums the values contained in the five-element integer array `a`. The `for` statement's body (line 14) does the totaling.

```
1  // fig06_04.c
2  // Computing the sum of the elements of an array.
3  #include <stdio.h>
4  #define SIZE 5
5
6  // function main begins program execution
7  int main(void) {
8     // use an initializer list to initialize the array
9     int a[SIZE] = {1, 2, 3, 4, 5};
10    int total = 0; // sum of array
11
12    // sum contents of array a
13    for (size_t i = 0; i < SIZE; ++i) {
14       total += a[i];
15    }
16
17    printf("The total of a's values is %d\n", total);
18 }
```

```
The total of a's values is 15
```

Fig. 6.4 | Computing the sum of the elements of an array.

6.4.5 Using Arrays to Summarize Survey Results

Our next example uses arrays to summarize the results of data collected in a survey. Consider the problem statement:

> *Twenty students were asked to rate the quality of the food in the student cafeteria on a scale of 1 to 5 (1 means awful, and 5 means excellent). Place the 20 responses in an integer array and summarize the results of the poll.*

Figure 6.5 is a typical array application. We wish to summarize the number of responses of each type. The 20-element array `responses` (lines 10–11) contains the students' responses. We use a 6-element array `frequency` (line 14) to count each response's number of occurrences. We ignore `frequency[0]` because it's logical to

have response 1 increment `frequency[1]` rather than `frequency[0]`. This allows us to use each response directly as the subscript in the `frequency` array. You should strive for program clarity. Sometimes it may be worthwhile to trade off the most efficient use of memory or processor time in favor of writing clearer programs. Sometimes performance considerations far outweigh clarity considerations.

PERF

```
1  // fig06_05.c
2  // Analyzing a student poll.
3  #include <stdio.h>
4  #define RESPONSES_SIZE 20 // define array sizes
5  #define FREQUENCY_SIZE 6
6
7  // function main begins program execution
8  int main(void) {
9     // place the survey responses in the responses array
10    int responses[RESPONSES_SIZE] =
11       {1, 2, 5, 4, 3, 5, 2, 1, 3, 1, 4, 3, 3, 3, 2, 3, 3, 2, 2, 5};
12
13    // initialize frequency counters to 0
14    int frequency[FREQUENCY_SIZE] = {0};
15
16    // for each answer, select the value of an element of the array
17    // responses and use that value as a subscript into the array
18    // frequency to determine the element to increment
19    for (size_t answer = 0; answer < RESPONSES_SIZE; ++answer) {
20       ++frequency[responses[answer]];
21    }
22
23    // display results
24    printf("%s%12s\n", "Rating", "Frequency");
25
26    // output the frequencies in a tabular format
27    for (size_t rating = 1; rating < FREQUENCY_SIZE; ++rating) {
28       printf("%6zu%12d\n", rating, frequency[rating]);
29    }
30 }
```

```
Rating   Frequency
     1           3
     2           5
     3           7
     4           2
     5           3
```

Fig. 6.5 | Analyzing a student poll.

How the `frequency` Counters Are Incremented

The `for` loop (lines 19–21) takes each response from `responses` and increments one of the five `frequency` array counters—`frequency[1]` to `frequency[5]`. The key statement in the loop is line 20:

```
++frequency[responses[answer]];
```

which increments the appropriate `frequency` counter, based on the value of the expression `responses[answer]`. When the counter variable `answer` is `0`, `responses[answer]` is `1`, so `++frequency[responses[answer]];` is interpreted as

```
++frequency[1];
```

which increments `frequency[1]`. When `answer` is `1`, the value of `responses[answer]` is `2`, so `++frequency[responses[answer]];` is interpreted as

```
++frequency[2];
```

which increments `frequency[2]`. When `answer` is `2`, the value of `responses[answer]` is `5`, so `++frequency[responses[answer]];` is interpreted as

```
++frequency[5];
```

which increments `frequency[5]`, and so on.

Invalid Survey Responses

Regardless of the number of survey responses processed, only a 6-element `frequency` array is required (ignoring element zero) to summarize the results. But what if the data contained an invalid value such as 13? In this case, the program would attempt to add `1` to `frequency[13]`, which is outside the array's bounds. C has no array bounds checking to prevent a program from referring to an element that does not exist. So, an executing program can "walk off" either end of an array without warning—a security problem we discuss in Section 6.13. Programs should validate that all input values are correct to prevent erroneous information from affecting a program's calculations.

SEC

Validate Array Subscripts

Referring to an element outside the array bounds is a logic error. When looping through an array, the array subscript should never go below 0 and should always be less than the total number of array elements—the array's size minus one. You should ensure that all array references remain within the bounds of the array.

ERR

6.4.6 Graphing Array Element Values with Bar Charts

Our next example (Fig. 6.6) reads numbers from an array and graphs the information in a bar chart. We display each number followed by a bar consisting of that many asterisks. The nested `for` statement (lines 17–19) displays the bars by iterating `n[i]` times and displaying one asterisk per iteration. Line 21 ends each bar.

```
1  // fig06_06.c
2  // Displaying a bar chart.
3  #include <stdio.h>
4  #define SIZE 5
5
6  // function main begins program execution
7  int main(void) {
```

Fig. 6.6 | Displaying a bar chart. (Part 1 of 2.)

```
   // use initializer list to initialize array n
   int n[SIZE] = {19, 3, 15, 7, 11};

   printf("%s%13s%17s\n", "Element", "Value", "Bar Chart");

   // for each element of array n, output a bar of the bar chart
   for (size_t i = 0; i < SIZE; ++i) {
      printf("%7zu%13d%8s", i, n[i], "");

      for (int j = 1; j <= n[i]; ++j) { // print one bar
         printf("%c", '*');
      }

      puts(""); // end a bar with a newline
   }
}
```

```
Element         Value         Bar Chart
      0            19         *******************
      1             3         ***
      2            15         ***************
      3             7         *******
      4            11         ***********
```

Fig. 6.6 | Displaying a bar chart. (Part 2 of 2.)

6.4.7 Rolling a Die 60,000,000 Times and Summarizing the Results in an Array

In Chapter 5, we stated that we'd show a more elegant way to write Fig. 5.5's dice-rolling program. Recall that the program rolled a single six-sided die 60,000,000 times and displayed the face counts. Figure 6.7 is an array version of Fig. 5.5. Line 17 replaces Fig. 5.5's entire 20-line `switch` statement. Once again, we use a `frequency` array in which we ignore element `0`, so we can use the face values as subscripts into the array.

```
1  // fig06_07.c
2  // Roll a six-sided die 60,000,000 times
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6  #define SIZE 7
7
8  // function main begins program execution
9  int main(void) {
10    srand(time(NULL)); // seed random number generator
11
12    int frequency[SIZE] = {0}; // initialize all frequency counts to 0
13
```

Fig. 6.7 | Roll a six-sided die 60,000,000 times. (Part 1 of 2.)

```
14     // roll die 60,000,000 times
15     for (int roll = 1; roll <= 60000000; ++roll) {
16        size_t face = 1 + rand() % 6;
17        ++frequency[face]; // replaces entire switch of Fig. 5.5
18     }
19
20     printf("%s%17s\n", "Face", "Frequency");
21
22     // output frequency elements 1-6 in tabular format
23     for (size_t face = 1; face < SIZE; ++face) {
24        printf("%4zu%17d\n", face, frequency[face]);
25     }
26  }
```

```
Face        Frequency
   1          9997167
   2         10003506
   3         10001940
   4          9995833
   5         10000843
   6         10000711
```

Fig. 6.7 | Roll a six-sided die 60,000,000 times. (Part 2 of 2.)

Self Check

1 *(Code)* Rewrite the following code segment to define seven-element `double` array `m` and initialize each of its elements to `10`:

```
int n[5]; // n is an array of five integers

// set elements of array n to 0
for (size_t i = 0; i < 5; ++i) {
   n[i] = 0; // set element at location i to 0
}
```

Answer:

```
double m[7]; // m is an array of 7 doubles

// set elements of array m to 10.0
for (size_t i = 0; i < 7; ++i) {
   m[i] = 10.0; // set element at location i to 10.0
}
```

2 *(Multiple Choice)* Which of the following statements a), b) or c) is *true*?

a) For an `int` array, if you provide fewer initializers than there are elements in the array, the remaining elements are initialized to `0`.

b) It's a syntax error to provide more initializers in an array initializer list than there are array elements—for example, `int n[3] = {32, 27, 64, 18};` is a syntax error, because there are four initializers but only three array elements.

c) If the array size is omitted from a definition with an initializer list, the compiler determines the number of elements based on the number of elements in the initializer list. So, the following creates the three-element `int` array `s`:

```
int s[] = {10, 20, 30};
```

d) All of the above statements are *true*.

Answer: d.

6.5 Using Character Arrays to Store and Manipulate Strings

Arrays can hold data of any type, though all the elements of a given array must have the same type. We now discuss storing strings in character arrays. So far, the only string-processing capability we have is outputting a string with `printf`. A string such as `"hello"` is really an array of individual characters, plus one more thing.

6.5.1 Initializing a Character Array with a String

Character arrays have several unique features. A character array can be initialized using a string literal. For example,

```
char string1[] = "first";
```

initializes the elements of array `string1` to the individual characters in the string literal `"first"`. In this case, the compiler determined array `string1`'s size based on the string's length. The string `"first"` contains five characters *plus a string-terminating null character*. So, `string1` actually contains six elements. The escape sequence representing the null character is `'\0'`. All strings end with this character. A character array representing a string should always be defined large enough to hold the string's number of characters and the terminating null character.

6.5.2 Initializing a Character Array with an Initializer List of Characters

Character arrays also can be initialized with individual character constants in an initializer list, but this can be tedious. The preceding definition is equivalent to

```
char string1[] = {'f', 'i', 'r', 's', 't', '\0'};
```

6.5.3 Accessing the Characters in a String

You can access a string's individual characters directly using array subscript notation. So, `string1[0]` is the character `'f'`, `string1[3]` is `'s'` and `string1[5]` is `'\0'`.

6.5.4 Inputting into a Character Array

The following definition creates a character array capable of storing a string of *at most 19 characters* and a *terminating null character*:

```
char string2[20];
```

The statement

```
scanf("%19s", string2);
```

reads a string from the keyboard into `string2`. You pass the array name to `scanf` without the `&` used with non-string variables. The `&` is normally used to provide `scanf` with a variable's *location* in memory so that a value can be stored there. In Section 6.7, when we discuss passing arrays to functions, we'll discuss why the `&` is not necessary for array names.

It's your responsibility to ensure that the array into which the string is read is capable of holding any string that the user types at the keyboard. Function `scanf` does *not* check how large the array is. It will read characters until a *space*, *tab*, *newline* or *end-of-file indicator* is encountered. The string `string2` should be no longer than 19 characters to leave room for the terminating null character. If the user types 20 or more characters, your program may crash or create a security vulnerability called a buffer overflow. For this reason, we used the conversion specification `%19s`. This tells `scanf` to read a maximum of 19 characters, preventing it from writing characters into memory beyond the end of `string2`. (In Section 6.13, we revisit the potential security issue raised by inputting into a character array and discuss the C standard's `scanf_s` function.)

SEC

6.5.5 Outputting a Character Array That Represents a String

A character array representing a string can be output with `printf` using the `%s` conversion specification. For example, you can print the character array `string2` with

```
printf("%s\n", string2);
```

Like `scanf`, `printf` does not check how large the character array is. It displays the string's characters until it encounters a terminating null character. [Consider what would print if, for some reason, the terminating null character were missing.]

6.5.6 Demonstrating Character Arrays

Figure 6.8 demonstrates initializing a character array with a string literal, reading a string into a character array, printing a character array as a string and accessing a string's individual characters. The program uses a `for` statement (lines 20–22) to loop through the `string1` array and print the individual characters separated by spaces, using the `%c` conversion specification. The condition in the `for` statement is true while the counter is less than the array's size and the terminating null character has not been encountered in the string. This program reads only strings that do not contain whitespace characters. We'll show how to read strings containing whitespace characters in Chapter 8.

```
1 // fig06_08.c
2 // Treating character arrays as strings.
3 #include <stdio.h>
4 #define SIZE 20
```

Fig. 6.8 | Treating character arrays as strings. (Part 1 of 2.)

```
5
6  // function main begins program execution
7  int main(void) {
8     char string1[SIZE] = ""; // reserves 20 characters
9     char string2[] = "string literal"; // reserves 15 characters
10
11    // prompt for string from user then read it into array string1
12    printf("%s", "Enter a string (no longer than 19 characters): ");
13    scanf("%19s", string1); // input no more than 19 characters
14
15    // output strings
16    printf("string1 is: %s\nstring2 is: %s\n", string1, string2);
17    puts("string1 with spaces between characters is:");
18
19    // output characters until null character is reached
20    for (size_t i = 0; i < SIZE && string1[i] != '\0'; ++i) {
21       printf("%c ", string1[i]);
22    }
23
24    puts("");
25 }
```

```
Enter a string (no longer than 19 characters): Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
H e l l o
```

Fig. 6.8 | Treating character arrays as strings. (Part 2 of 2.)

Self Check

1 *(Multiple Choice)* Which of the following statements a), b) or c) is *false*?

a) A character array representing a string can be output with `printf` and the `%s` conversion specifier, as in:

```
printf("%s\n", month);
```

b) Function `printf`, like `scanf`, does not check how large the character array is.

c) When function `printf` displays the characters of a character array representing a string, it stops when it tries to print the first character past the end of the array.

d) All of the above statements are *true*.

Answer: c) is *false*. Actually, `printf` keeps displaying characters until it encounters a terminating null character, even if that is well beyond the end of the array.

2 *(True/False)* The following array can store a string of at most 20 characters and a terminating null character:

```
char name1[20];
```

Answer: *False*. Actually, the statement creates a character array capable of storing a string of at most 19 characters and a terminating null character.

6.6 Static Local Arrays and Automatic Local Arrays

Chapter 5 discussed the storage-class specifier `static`. A `static` local variable exists for the program's duration but is visible only in the function body. We can apply `static` to a local array definition to prevent the array from being created and initialized every time the function is called and destroyed every time the function exits. This reduces program execution time, particularly for programs with frequently called functions that contain large arrays. Arrays that are `static` are initialized once at program startup. If you do not explicitly initialize a `static` array, that array's elements are initialized to zero by default.

PERF

Figure 6.9 demonstrates function `staticArrayInit` (lines 21–38) with a local `static` array (line 23) and function `automaticArrayInit` (lines 41–58) with a local automatic array (line 43). Function `staticArrayInit` is called twice (lines 11 and 15). The local `static` array in the function is initialized to zero at program startup (line 23). The function prints the array, adds 5 to each element and prints the array again. The second time the function is called, the `static` array contains the values stored during the first call.

```
1  // fig06_09.c
2  // Static arrays are initialized to zero if not explicitly initialized.
3  #include <stdio.h>
4
5  void staticArrayInit(void); // function prototype
6  void automaticArrayInit(void); // function prototype
7
8  // function main begins program execution
9  int main(void) {
10    puts("First call to each function:");
11    staticArrayInit();
12    automaticArrayInit();
13
14    puts("\n\nSecond call to each function:");
15    staticArrayInit();
16    automaticArrayInit();
17    puts("");
18 }
19
20 // function to demonstrate a static local array
21 void staticArrayInit(void) {
22    // initializes elements to 0 before the function is called
23    static int array1[3];
24
25    puts("\nValues on entering staticArrayInit:");
26
27    // output contents of array1
28    for (size_t i = 0; i <= 2; ++i) {
29       printf("array1[%zu] = %d  ", i, array1[i]);
30    }
```

Fig. 6.9 | Static arrays are initialized to zero if not explicitly initialized. (Part 1 of 2.)

```
31
32     puts("\nValues on exiting staticArrayInit:");
33
34     // modify and output contents of array1
35     for (size_t i = 0; i <= 2; ++i) {
36        printf("array1[%zu] = %d  ", i, array1[i] += 5);
37     }
38  }
39
40  // function to demonstrate an automatic local array
41  void automaticArrayInit(void) {
42     // initializes elements each time function is called
43     int array2[3] = {1, 2, 3};
44
45     puts("\n\nValues on entering automaticArrayInit:");
46
47     // output contents of array2
48     for (size_t i = 0; i <= 2; ++i) {
49        printf("array2[%zu] = %d  ", i, array2[i]);
50     }
51
52     puts("\nValues on exiting automaticArrayInit:");
53
54     // modify and output contents of array2
55     for (size_t i = 0; i <= 2; ++i) {
56        printf("array2[%zu] = %d  ", i, array2[i] += 5);
57     }
58  }
```

```
First call to each function:

Values on entering staticArrayInit:
array1[0] = 0  array1[1] = 0  array1[2] = 0
Values on exiting staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8

Second call to each function:

Values on entering staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5 — values preserved from last call
Values on exiting staticArrayInit:
array1[0] = 10  array1[1] = 10  array1[2] = 10

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3 — values reinitialized after last call
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8
```

Fig. 6.9 | Static arrays are initialized to zero if not explicitly initialized. (Part 2 of 2.)

Function `automaticArrayInit` is also called twice (lines 12 and 16). The automatic local array's elements are initialized with the values 1, 2 and 3 (line 43). The function prints the array, adds 5 to each element and prints the array again. The second time the function is called, the array elements are initialized to 1, 2 and 3 again because the array has automatic storage duration

Self Check

1 *(Multiple Choice)* Which of the following statements a), b) or c) is *false*?
   a) A `static` local variable exists for the duration of the program but is visible only in the function body.
   b) A `static` local array is created and initialized once, not each time the function is called. This reduces program execution time, particularly for programs with frequently called functions that contain large arrays.
   c) If you do not explicitly initialize a `static` array, that array's elements are initialized to zero by default.
   d) All of the above statements are *true*.

Answer: d.

6.7 Passing Arrays to Functions

To pass an array argument to a function, specify the array's name without any brackets. For example, if array `hourlyTemperatures` has been defined as

```
int hourlyTemperatures[HOURS_IN_A_DAY];
```

the function call

```
modifyArray(hourlyTemperatures, HOURS_IN_A_DAY)
```

passes array `hourlyTemperatures` and its size to function `modifyArray`.

Recall that all arguments in C are passed by value. However, C automatically passes arrays to functions by reference—the called functions can modify the callers' original array element values. We'll see in Chapter 7 that this is not a contradiction. An array's name evaluates to the address in memory of the array's first element. Because the array's starting address is passed, the called function knows precisely where the array is stored. So, any modifications the called function makes to the array elements change the values of the original array's elements in the caller.

Showing That an Array Name Is an Address

Figure 6.10 demonstrates that "the value of an array name" is really the *address* of the array's first element by printing `array`, `&array[0]` and `&array` using the **`%p` conversion specification** for printing addresses. The `%p` conversion specification normally outputs addresses as hexadecimal numbers, but this is compiler-dependent. Hexadecimal (base 16) numbers consist of the digits 0 through 9 and the letters A through F—the hexadecimal equivalents of the decimal numbers 10–15. Online Appendix E provides an in-depth discussion of the relationships among binary (base 2), octal (base 8), decimal (base 10; standard integers) and hexadecimal integers. The output shows

that `array`, `&array` and `&array[0]` have the same value. This program's output is system dependent, but the addresses are always identical for each program execution on a particular computer.

```
1  // fig06_10.c
2  // Array name is the same as the address of the array's first element.
3  #include <stdio.h>
4
5  // function main begins program execution
6  int main(void) {
7     char array[5] = ""; // define an array of size 5
8
9     printf("    array = %p\n&array[0] = %p\n   &array = %p\n",
10       array, &array[0], &array);
11 }
```

```
    array = 0031F930
&array[0] = 0031F930
   &array = 0031F930
```

Fig. 6.10 | Array name is the same as the address of the array's first element.

Passing arrays by reference makes sense for performance reasons. If arrays were passed by value, a copy of each element would be passed. For large, frequently passed arrays, this would be time-consuming and would consume storage for the copies of the arrays. It's possible to pass an array by value (by placing it in a `struct`, as we explain in Chapter 10, Structures, Unions, Bit Manipulation and Enumerations).

PERF

SE

Passing Individual Array Elements

Although entire arrays are passed by reference, individual array elements are passed by value, exactly as any other variable. Single pieces of data, such as individual `int`s, `float`s and `char`s, are called **scalars**. To pass an array element to a function, use the subscripted array name as an argument in the function call. In Chapter 7, we show how to pass scalars (i.e., individual variables and array elements) to functions by reference.

Array Parameters

For a function to receive an array through a function call, the function's parameter list must expect an array. The function header for function `modifyArray` (from earlier in this section) can be written as

```
void modifyArray(int b[], size_t size)
```

indicating that `modifyArray` expects to receive an `int` array in parameter `b` and the array's number of elements in parameter `size`. The array's number of elements is not required in the array parameter's brackets. If it's included, the compiler checks that it's greater than zero, then ignores it—specifying a negative value is a compilation error. When the called function uses the array name `b`, it will be referring to the caller's original array. So, in the function call:

ERR

```
modifyArray(hourlyTemperatures, HOURS_IN_A_DAY)
```

`modifyArray`'s parameter `b` represents `hourlyTemperatures` in the caller. In Chapter 7, we introduce other notations for indicating that a function receives an array. As we'll see, these notations are based on the intimate relationship between arrays and pointers.

Difference Between Passing an Entire Array and Passing an Array Element

Figure 6.11 demonstrates the difference between passing an entire array and passing an individual array element. The program first prints integer array `a`'s five elements (lines 18–20). Next, we pass array `a` and its size to `modifyArray` (line 24), which multiplies each of `a`'s elements by 2 (lines 45–47). Then, lines 28–30 display `a`'s updated contents. As the output shows, `modifyArray` did, indeed, modify `a`'s elements. Next, line 34 prints `a[3]`'s value and line 36 passes it to function `modifyElement`. The function multiplies its argument by 2 (line 53) and prints the new value. When line 39 in `main` displays `a[3]` again, it has *not* been modified because individual array elements are passed by value.

```
1  // fig06_11.c
2  // Passing arrays and individual array elements to functions.
3  #include <stdio.h>
4  #define SIZE 5
5
6  // function prototypes
7  void modifyArray(int b[], size_t size);
8  void modifyElement(int e);
9
10 // function main begins program execution
11 int main(void) {
12    int a[SIZE] = {0, 1, 2, 3, 4}; // initialize array a
13
14    puts("Effects of passing entire array by reference:\n\nThe "
15       "values of the original array are:");
16
17    // output original array
18    for (size_t i = 0; i < SIZE; ++i) {
19       printf("%3d", a[i]);
20    }
21
22    puts(""); // outputs a newline
23
24    modifyArray(a, SIZE); // pass array a to modifyArray by reference
25    puts("The values of the modified array are:");
26
27    // output modified array
28    for (size_t i = 0; i < SIZE; ++i) {
29       printf("%3d", a[i]);
30    }
31
```

Fig. 6.11 | Passing arrays and individual array elements to functions. (Part 1 of 2.)

```
32    // output value of a[3]
33    printf("\n\n\nEffects of passing array element "
34       "by value:\n\nThe value of a[3] is %d\n", a[3]);
35
36    modifyElement(a[3]); // pass array element a[3] by value
37
38    // output value of a[3]
39    printf("The value of a[3] is %d\n", a[3]);
40 }
41
42 // in function modifyArray, "b" points to the original array "a" in memory
43 void modifyArray(int b[], size_t size) {
44    // multiply each array element by 2
45    for (size_t j = 0; j < size; ++j) {
46       b[j] *= 2; // actually modifies original array
47    }
48 }
49
50 // in function modifyElement, "e" is a local copy of array element
51 // a[3] passed from main
52 void modifyElement(int e) {
53    e *= 2; // multiply parameter by 2
54    printf("Value in modifyElement is %d\n", e);
55 }
```

```
Effects of passing entire array by reference:

The values of the original array are:
  0  1  2  3  4
The values of the modified array are:
  0  2  4  6  8


Effects of passing array element by value:

The value of a[3] is 6
Value in modifyElement is 12
The value of a[3] is 6
```

Fig. 6.11 | Passing arrays and individual array elements to functions. (Part 2 of 2.)

Using `const` to Prevent Functions from Modifying Array Elements

There may be situations in your programs in which a function should *not* modify array elements. C's type qualifier **`const`** (short for "constant") can prevent a function from modifying an argument. When an array parameter is preceded by the `const` qualifier, the function treats the array elements as constants. Any attempt to modify an array element in the function body results in a compile-time error. This is another example of the principle of least privilege. A function should not be given the capability to modify an array in the caller unless it's absolutely necessary.

SE

ERR

The following definition of a function named `tryToModifyArray` uses the parameter `const int b[]` (line 3) to specify that array `b` is constant and cannot be modified:

```
1  // in function tryToModifyArray, array b is const, so it cannot be
2  // used to modify its array argument in the caller
3  void tryToModifyArray(const int b[]) {
4     b[0] /= 2; // error
5     b[1] /= 2; // error
6     b[2] /= 2; // error
7  }
```

Each of the function's attempts to modify array elements results in a compiler error. The `const` qualifier is discussed in additional contexts in Chapter 7.

Self Check

1 *(Multiple Choice)* Which of the following statements is *false*?

a) Given the following array `hourlyTemperatures`:

```
int hourlyTemperatures[HOURS_IN_A_DAY];
```

the following function call passes `hourlyTemperatures` and its size to `modifyArray`:

```
modifyArray(hourlyTemperatures, HOURS_IN_A_DAY)
```

b) Recall that all arguments in C are passed by value, so C automatically passes arrays to functions by value.

c) The array's name evaluates to the address of the array's first element.

d) Because the address of the array's first element is passed, the called function knows precisely where the array is stored.

Answer: b) is *false*. Actually, C automatically passes arrays to functions by reference—the called functions can modify the element values in the callers' original arrays.

2 *(Discussion)* Based on the meaningful function name and parameter definitions in the following function header, describe as much as you can about what this function probably does:

```
void modifyArray(int b[], size_t size)
```

Answer: Function `modifyArray` expects to receive an array of integers in parameter `b` and the number of array elements in parameter `size`. The array is passed in by reference, so the function is able to modify the original array in the caller.

6.8 Sorting Arrays

Sorting data—that is, placing the data into ascending or descending order—is one of the most important computing applications. A bank sorts checks by account number to prepare individual bank statements at the end of each month. Telephone companies sort their lists of accounts by last name and, within that, by first name to make it easy to find phone numbers. Virtually every organization must sort some data, and in many cases, massive amounts of it. Sorting data is an intriguing problem that has attracted some of the most intense computer-science research efforts. Here, we dis-

cuss a simple sorting scheme. In Chapters 12 and 13, we investigate more complex schemes that yield better performance. Often, the simplest algorithms perform poorly. Their virtue is that they're easy to write, test and debug. More complex algorithms are often needed to realize maximum performance.

PERF

Bubble Sort

Figure 6.12 sorts the 10-element array a's values (line 8) into ascending order. The technique we use is called the **bubble sort** or the **sinking sort** because the smaller values gradually "bubble" their way to the top of the array like air bubbles rising in water, while the larger values sink to the bottom of the array. The technique uses several passes through the array. On each pass, the algorithm compares successive pairs of elements (element 0 and element 1, then element 1 and element 2, etc.). If a pair is in increasing order (or if the values are identical), we leave the values as they are. If a pair is in decreasing order, we swap their values in the array.

```
1  // fig06_12.c
2  // Sorting an array's values into ascending order.
3  #include <stdio.h>
4  #define SIZE 10
5
6  // function main begins program execution
7  int main(void) {
8     int a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};
9
10    puts("Data items in original order");
11
12    // output original array
13    for (size_t i = 0; i < SIZE; ++i) {
14       printf("%4d", a[i]);
15    }
16
17    // bubble sort
18    // loop to control number of passes
19    for (int pass = 1; pass < SIZE; ++pass) {
20       // loop to control number of comparisons per pass
21       for (size_t i = 0; i < SIZE - 1; ++i) {
22          // compare adjacent elements and swap them if first
23          // element is greater than second element
24          if (a[i] > a[i + 1]) {
25             int hold = a[i];
26             a[i] = a[i + 1];
27             a[i + 1] = hold;
28          }
29       }
30    }
31
32    puts("\nData items in ascending order");
33
```

Fig. 6.12 | Sorting an array's values into ascending order. (Part 1 of 2.)

```
34      // output sorted array
35      for (size_t i = 0; i < SIZE; ++i) {
36         printf("%4d", a[i]);
37      }
38
39      puts("");
40   }
```

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

Fig. 6.12 | Sorting an array's values into ascending order. (Part 2 of 2.)

First, the program compares `a[0]` to `a[1]`, then `a[1]` to `a[2]`, then `a[2]` to `a[3]`, and so on until it completes the pass by comparing `a[8]` to `a[9]`. Although there are 10 elements, only nine comparisons are performed. A large value may move down the array many positions on a single pass because of how the successive comparisons are made, but a small value may move up only one position.

On the first pass, the largest value is guaranteed to sink to the array's bottom element, `a[9]`. On the second pass, the second-largest value is guaranteed to sink to `a[8]`. On the ninth pass, the ninth-largest value sinks to `a[1]`. This leaves the smallest value in `a[0]`, so only nine passes are needed to sort the 10-element array.

Swapping Elements

The sorting is performed by the nested `for` loops (lines 19–30). If a swap is necessary, it's performed by the three assignments in lines 25–27

```
int hold = a[i];
a[i] = a[i + 1];
a[i + 1] = hold;
```

The variable `hold` temporarily stores one of the two values being swapped. The swap cannot be performed with only the two assignments

```
a[i] = a[i + 1];
a[i + 1] = a[i];
```

If, for example, `a[i]` is `7` and `a[i + 1]` is `5`, after the first assignment, both values will be `5` and the value `7` will be lost—hence the need for the extra variable `hold`.

Bubble Sort Is Easy to Implement, But Slow

The chief virtue of the bubble sort is that it's easy to program. However, it runs slowly because every exchange moves an element only one position closer to its final destination. This becomes apparent when sorting large arrays. In the exercises, we'll develop more efficient versions of the bubble sort. Far more efficient sorts than the bubble sort have been developed. We'll investigate other algorithms in Chapter 13. More advanced courses investigate sorting and searching in greater depth.

Self Check

1 *(Multiple Choice)* Which of the following statements a), b) or c) is *true*?
   a) The chief virtue of the bubble sort is that it's easy to program.
   b) Bubble sort runs slowly because every exchange moves an element only one position closer to its final position. This becomes apparent when sorting large arrays.
   c) Far more efficient sorts than the bubble sort have been developed.
   d) All of the above statements are *true*.

Answer: d.

2 *(True/False)* If a swap is necessary in the bubble sort, it uses the assignments:

```
a[i] = a[i + 1];
a[i + 1] = a[i];
```

Answer: *False*. Actually, the swap requires one more variable and one more statement:

```
int hold = a[i];
a[i] = a[i + 1];
a[i + 1] = hold;
```

where the variable `hold` temporarily stores one of the two values being swapped.

6.9 Intro to Data Science Case Study: Survey Data Analysis

We now consider a larger example. Computers are commonly used for **survey data analysis** to compile and analyze the results of surveys and opinion polls. Figure 6.13 uses the array `response` initialized with 99 responses to a survey. Each response is a number from 1 to 9. The program computes the mean, median and mode of the 99 values. This example includes many common manipulations required in array problems, including passing arrays to functions. Notice that lines 48–52 contain several string literals separated only by whitespace. C compilers automatically combine such string literals into one—this helps making long string literals more readable.

```
1  // fig06_13.c
2  // Survey data analysis with arrays:
3  // computing the mean, median and mode of the data.
4  #include <stdio.h>
5  #define SIZE 99
6
7  // function prototypes
8  void mean(const int answer[]);
9  void median(int answer[]);
10 void mode(int freq[], const int answer[]) ;
11 void bubbleSort(int a[]);
12 void printArray(const int a[]);
```

Fig. 6.13 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 1 of 5.)

```
13
14 // function main begins program execution
15 int main(void) {
16    int frequency[10] = {0}; // initialize array frequency
17
18    // initialize array response
19    int response[SIZE] =
20       {6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
21        7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
22        6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
23        7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
24        6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
25        7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
26        5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
27        7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
28        7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
29        4, 5, 6, 1, 6, 5, 7, 8, 7};
30
31    // process responses
32    mean(response);
33    median(response);
34    mode(frequency, response);
35 }
36
37 // calculate average of all response values
38 void mean(const int answer[]) {
39    printf("%s\n%s\n%s\n", "--------", "  Mean", "--------");
40
41    int total = 0; // variable to hold sum of array elements
42
43    // total response values
44    for (size_t j = 0; j < SIZE; ++j) {
45       total += answer[j];
46    }
47
48    printf("The mean is the average value of the data\n"
49           "items. The mean is equal to the total of\n"
50           "all the data items divided by the number\n"
51           "of data items (%u). The mean value for\n"
52           "this run is: %u / %u = %.4f\n\n",
53           SIZE, total, SIZE, (double) total / SIZE);
54 }
55
56 // sort array and determine median element's value
57 void median(int answer[]) {
58    printf("\n%s\n%s\n%s\n%s", "--------", " Median", "--------",
59           "The unsorted array of responses is");
60
61    printArray(answer); // output unsorted array
62
63    bubbleSort(answer); // sort array
```

Fig. 6.13 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 2 of 5.)

```
64
65     printf("%s", "\n\nThe sorted array is");
66     printArray(answer); // output sorted array
67
68     // display median element
69     printf("\n\nThe median is element %u of\n"
70            "the sorted %u element array.\n"
71            "For this run the median is %u\n\n\n",
72            SIZE / 2, SIZE, answer[SIZE / 2]);
73  }
74
75  // determine most frequent response
76  void mode(int freq[], const int answer[]) {
77     printf("\n%s\n%s\n%s\n", "--------", "  Mode", "--------");
78
79     // initialize frequencies to 0
80     for (size_t rating = 1; rating <= 9; ++rating) {
81        freq[rating] = 0;
82     }
83
84     // summarize frequencies
85     for (size_t j = 0; j < SIZE; ++j) {
86        ++freq[answer[j]];
87     }
88
89     // output headers for result columns
90     printf("%s%11s%19s\n\n%54s\n%54s\n\n",
91            "Response", "Frequency", "Bar Chart",
92            "1    1    2    2", "5    0    5    0    5");
93
94     // output results
95     int largest = 0; // represents largest frequency
96     int modeValue = 0; // represents most frequent response
97
98     for (size_t rating = 1; rating <= 9; ++rating) {
99        printf("%8zu%11d          ", rating, freq[rating]);
100
101       // keep track of mode value and largest frequency value
102       if (freq[rating] > largest) {
103          largest = freq[rating];
104          modeValue = rating;
105       }
106
107       // output bar representing frequency value
108       for (int h = 1; h <= freq[rating]; ++h) {
109          printf("%s", "*");
110       }
111
112       puts(""); // being new line of output
113    }
114
```

Fig. 6.13 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 3 of 5.)

```
   // display the mode value
   printf("\nThe mode is the most frequent value.\n"
          "For this run the mode is %d which occurred %d times.\n",
          modeValue, largest);
}

// function that sorts an array with bubble sort algorithm
void bubbleSort(int a[]) {
   // loop to control number of passes
   for (int pass = 1; pass < SIZE; ++pass) {
      // loop to control number of comparisons per pass
      for (size_t j = 0; j < SIZE - 1; ++j) {
         // swap elements if out of order
         if (a[j] > a[j + 1]) {
            int hold = a[j];
            a[j] = a[j + 1];
            a[j + 1] = hold;
         }
      }
   }
}

// output array contents (20 values per row)
void printArray(const int a[]) {
   // output array contents
   for (size_t j = 0; j < SIZE; ++j) {

      if (j % 20 == 0) { // begin new line every 20 values
         puts("");
      }

      printf("%2d", a[j]);
   }
}
```

```
--------
  Mean
--------
The mean is the average value of the data
items. The mean is equal to the total of
all the data items divided by the number
of data items (99). The mean value for
this run is: 681 / 99 = 6.8788


--------
 Median
--------
The unsorted array of responses is
 6 7 8 9 8 7 8 9 8 9 7 8 9 5 9 8 7 8 7 8
 6 7 8 9 3 9 8 7 8 7 7 8 9 8 9 8 9 7 8 9
```

Fig. 6.13 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 4 of 5.)

```
 6 7 8 7 8 7 9 8 9 2 7 8 9 8 9 8 9 7 5 3
 5 6 7 2 5 3 9 4 6 4 7 8 9 6 8 7 8 9 7 8
 7 4 4 2 5 3 8 7 5 6 4 5 6 1 6 5 7 8 7

The sorted array is
 1 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5 5 5 5
 5 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7
 7 7 7 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8
 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9

The median is element 49 of
the sorted 99 element array.
For this run the median is 7


--------
  Mode
--------
Response  Frequency          Bar Chart

                                    1    1    2    2
                               5    0    5    0    5

       1          1          *
       2          3          ***
       3          4          ****
       4          5          *****
       5          8          ********
       6          9          *********
       7         23          ***********************
       8         27          ***************************
       9         19          *******************

The mode is the most frequent value.
For this run the mode is 8 which occurred 27 times.
```

Fig. 6.13 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 5 of 5.)

Mean

The mean is the arithmetic average of the 99 values. Function `mean` (lines 38–54) computes the mean by totaling the 99 elements and dividing the result by 99.

Median

The median is the middle value. Function `median` (lines 57–73) first sorts the responses by calling function `bubbleSort` (defined in lines 122–135). Then it determines the median by picking the sorted array's middle element, `answer[SIZE / 2]`. When the number of elements is even, the median should be calculated as the mean of the two middle elements—function `median` does not currently provide this capability. Lines 61 and 66 call function `printArray` (lines 138–148) to output the `response` array before and after the sort.

Mode

The mode is the value that occurs most frequently among the 99 responses. Function `mode` (lines 76–119) determines the mode by counting the number of responses of each type, then selecting the value with the greatest count. This version of function `mode` does not handle a tie (see Exercise 6.14). Function `mode` also produces a bar chart to aid in determining the mode graphically.

Related Exercises

This case study is supported by the following exercise:

- Exercise 6.14 (Mean, Median and Mode Program Modifications).

Self Check

1 *(Multiple Choice)* Which of the following statements a), b) or c) is *false*?

a) The median is the middle value among the sorted data items.
b) The algorithm for finding the median of the values in an array sorts the array into ascending order, then picks the sorted array's middle element.
c) When the number of elements is even, the median is calculated as the mean of the two middle elements.
d) All of the above statements are *true*.

Answer: d.

2 *(Multiple Choice)* Which of the following statements is a), b) or c) is *false*?

a) The mode is the value that occurs most frequently among the data items.
b) The algorithm for finding the mode counts the number of occurrences of each value, then selects the most frequently occurring value. One problem with determining the `mode` is what to do in case of a tie.
c) The `mode` can be determined visually by graphing the frequencies of the values in a bar chart—the longest bar represents the mode.
d) All of the above statements are *true*.

Answer: d.

6.10 Searching Arrays

You'll often work with large amounts of data stored in arrays. It may be necessary to determine whether an array contains a value that matches a certain **key value**. The process of finding a key value in an array is called **searching**. This section discusses two searching techniques—the simple **linear search** technique and the more efficient (but more complicated) **binary search** technique. Exercises 6.32 and 6.33 ask you to implement recursive versions of the linear search and the binary search.

6.10.1 Searching an Array with Linear Search

A linear search (Fig. 6.14, lines 37–46) compares each array element with the **search key**. The array is not sorted, so it's just as likely the value will be found in the first element as in the last. On average, therefore, the program will have to compare the

search key with *half* the array elements. If the key value is found, we return the element's subscript; otherwise, we return -1 (an invalid subscript).

```c
// fig06_14.c
// Linear search of an array.
#include <stdio.h>
#define SIZE 100

// function prototype
int linearSearch(const int array[], int key, size_t size);

// function main begins program execution
int main(void) {
   int a[SIZE] = {0}; // create array a

   // create some data
   for (size_t x = 0; x < SIZE; ++x) {
      a[x] = 2 * x;
   }

   printf("Enter integer search key: ");
   int searchKey = 0; // value to locate in array a
   scanf("%d", &searchKey);

   // attempt to locate searchKey in array a
   int subscript = linearSearch(a, searchKey, SIZE);

   // display results
   if (subscript != -1) {
      printf("Found value at subscript %d\n", subscript);
   }
   else {
      puts("Value not found");
   }
}

// compare key to every element of array until the location is found
// or until the end of array is reached; return subscript of element
// if key is found or -1 if key is not found
int linearSearch(const int array[], int key, size_t size) {
   // loop through array
   for (size_t n = 0; n < size; ++n) {
      if (array[n] == key) {
         return n; // return location of key
      }
   }

   return -1; // key not found
}
```

```
Enter integer search key: 36
Found value at subscript 18
```

Fig. 6.14 | Linear search of an array. (Part 1 of 2.)

```
Enter integer search key: 37
Value not found
```

Fig. 6.14 | Linear search of an array. (Part 2 of 2.)

6.10.2 Searching an Array with Binary Search

Linear searching works well for small or unsorted arrays. However, for large arrays, linear searching is inefficient. If the array is sorted, the high-speed binary search technique can be used.

The binary search algorithm eliminates from consideration *one-half* of a sorted array's elements after each comparison. The algorithm locates the middle array element and compares it to the search key. If they're equal, the algorithm found the search key, so it returns that element's subscript. If they're not equal, the problem is reduced to searching *one-half* of the array. If the search key is less than the middle array element, the algorithm searches the *first half* of the array; otherwise, it searches the *second half*. If the search key is not the middle element in the current subarray (a piece of the original array), the algorithm repeats on one-quarter of the original array. The search continues until the search key is equal to the middle element of a subarray, or until the subarray consists of one element that's not equal to the search key—that is, the search key is not found.

PERF

Performance of the Binary Search Algorithm

In the worst-case scenario, searching a sorted array of 1023 elements takes only 10 comparisons using a binary search. Repeatedly dividing 1,024 by 2 yields the values 512, 256, 128, 64, 32, 16, 8, 4, 2 and 1. The number 1,024 (2^{10}) is divided by 2 only 10 times to get the value 1. Dividing by 2 is equivalent to one comparison in the binary search algorithm. An array of 1,048,576 (2^{20}) elements takes a maximum of only 20 comparisons to find the search key. A sorted array of one billion elements takes a maximum of only 30 comparisons to find the search key. This is a tremendous increase in performance over a linear search of a sorted array, which requires comparing the search key to an average of half of the array elements. For a one-billion-element array, this is a difference between an average of 500 million comparisons and a maximum of 30 comparisons! The maximum comparisons for any array can be determined by finding the first power of 2 greater than the number of array elements.

Implementing Binary Search

Figure 6.15 presents the iterative version of function `binarySearch` (lines 39–60). The function receives four arguments—an integer array `b` to search, an integer `key` to find, the `low` array subscript and the `high` array subscript. The last two arguments define the portion of the array to search. If the search key does not match the middle element of a subarray, the `low` subscript or `high` subscript is modified so that a smaller subarray can be searched:

- If the search key is less than the middle element, the algorithm sets the `high` subscript to `middle - 1` (line 52), then continues the search on the elements with subscripts in the range `low` to `middle - 1`.
- If the search key is greater than the middle element, the algorithm sets the `low` subscript to `middle + 1` (line 55), then continues the search on the elements with subscripts in the range `middle + 1` to `high`.

The program uses an array of 15 elements. The first power of 2 greater than the number of elements in this array is 16 (2^4), so no more than 4 comparisons are required to find the search key. We use function `printHeader` (lines 63–79) to output the array subscripts and function `printRow` (lines 83–99) to output each subarray during the binary search process. We mark the middle element in each subarray with an asterisk (*) to indicate the element to which the search key is compared.

```
1  // fig06_15.c
2  // Binary search of a sorted array.
3  #include <stdio.h>
4  #define SIZE 15
5
6  // function prototypes
7  int binarySearch(const int b[], int key, size_t low, size_t high);
8  void printHeader(void);
9  void printRow(const int b[], size_t low, size_t mid, size_t high);
10
11 // function main begins program execution
12 int main(void) {
13    int a[SIZE] = {0}; // create array a
14
15    // create data
16    for (size_t i = 0; i < SIZE; ++i) {
17       a[i] = 2 * i;
18    }
19
20    printf("%s", "Enter a number between 0 and 28: ");
21    int key = 0; // value to locate in array a
22    scanf("%d", &key);
23
24    printHeader();
25
26    // search for key in array a
27    int result = binarySearch(a, key, 0, SIZE - 1);
28
29    // display results
30    if (result != -1) {
31       printf("\n%d found at subscript %d\n", key, result);
32    }
33    else {
34       printf("\n%d not found\n", key);
35    }
36 }
```

Fig. 6.15 | Binary search of a sorted array. (Part 1 of 3.)

```
37
38 // function to perform binary search of an array
39 int binarySearch(const int b[], int key, size_t low, size_t high) {
40    // loop until low subscript is greater than high subscript
41    while (low <= high) {
42       size_t middle = (low + high) / 2; // determine middle subscript
43
44       // display subarray used in this loop iteration
45       printRow(b, low, middle, high);
46
47       // if key matches, return middle subscript
48       if (key == b[middle]) {
49          return middle;
50       }
51       else if (key < b[middle]) { // if key < b[middle], adjust high
52          high = middle - 1; // next iteration searches low end of array
53       }
54       else { // key > b[middle], so adjust low
55          low = middle + 1; // next iteration searches high end of array
56       }
57    } // end while
58
59    return -1; // searchKey not found
60 }
61
62 // Print a header for the output
63 void printHeader(void) {
64    puts("\nSubscripts:");
65
66    // output column head
67    for (int i = 0; i < SIZE; ++i) {
68       printf("%3d ", i);
69    }
70
71    puts(""); // start new line of output
72
73    // output line of - characters
74    for (int i = 1; i <= 4 * SIZE; ++i) {
75       printf("%s", "-");
76    }
77
78    puts(""); // start new line of output
79 }
80
81 // Print one row of output showing the current
82 // part of the array being processed.
83 void printRow(const int b[], size_t low, size_t mid, size_t high) {
84    // loop through entire array
85    for (size_t i = 0; i < SIZE; ++i) {
86       // display spaces if outside current subarray range
87       if (i < low || i > high) {
88          printf("%s", "    ");
89       }
```

Fig. 6.15 | Binary search of a sorted array. (Part 2 of 3.)

```
90         else if (i == mid) { // display middle element
91            printf("%3d*", b[i]); // mark middle value
92         }
93         else { // display other elements in subarray
94            printf("%3d ", b[i]);
95         }
96      }
97
98      puts(""); // start new line of output
99   }
```

```
Enter a number between 0 and 28: 25

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
-------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
                               16  18  20  22* 24  26  28
                                           24  26* 28
                                           24*

25 not found
```

```
Enter a number between 0 and 28: 8

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
-------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12
                  8  10* 12
                  8*

8 found at subscript 4
```

```
Enter a number between 0 and 28: 6

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
-------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12

6 found at subscript 3
```

Fig. 6.15 | Binary search of a sorted array. (Part 3 of 3.)

Self Check

1 *(Multiple Choice)* Which of the following statements about linear search is *false*?

a) It compares every array element with the search key.

b) Because the array is not in any particular order, it's just as likely that the value will be found in the first element as in the last.

c) On average, it compares the search key with half the array's elements.

d) It works well for small or unsorted arrays. For *large* arrays it's *inefficient*.

Answer: a) is *false*. It could find a match before reaching the array's end, in which case it would terminate the search before comparing every element with the search key.

2 ***(Multiple Choice)*** Which of the following statements about binary search is *false*?

a) If the array is sorted, the high-speed binary search technique can be used.

b) The binary search algorithm eliminates from consideration two of the elements in a sorted array after each comparison.

c) The algorithm locates the middle element of the array and compares it to the search key. If they're equal, the search key is found and the array index of that element is returned. If the search key is less than the middle element of the array, the algorithm then searches the array's first half; otherwise, the algorithm searches the array's second half.

d) The search continues until the search key is equal to a subarray's middle element, or until the subarray consists of one element that's not equal to the search key (i.e., the search key is not found).

Answer: b) is *false*. Actually, the binary search algorithm eliminates from consideration one-half of the elements in a sorted array after each comparison.

6.11 Multidimensional Arrays

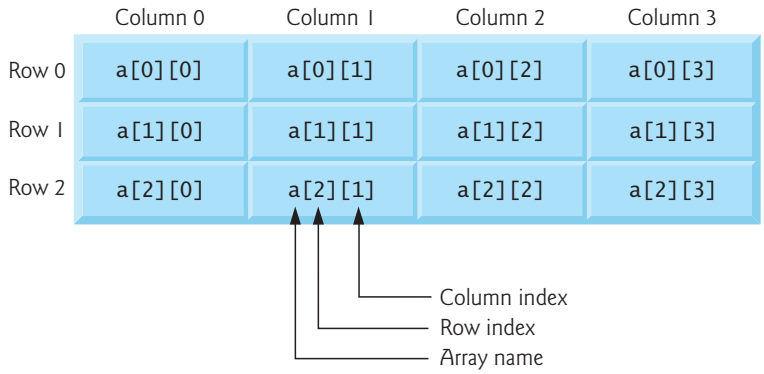
Arrays can have multiple subscripts. A common use of **multidimensional arrays** is to represent tables of values consisting of information arranged in *rows* and *columns*. To identify a particular table element, we specify two subscripts:

- The *first* (by convention) identifies the element's *row* and
- the *second* (by convention) identifies the element's *column*.

Arrays that require two subscripts to identify a particular element commonly are called **two-dimensional arrays**. Multidimensional arrays can have more than two subscripts.

6.11.1 Illustrating a Two-Dimensional Array

The following diagram illustrates a two-dimensional array named a:



The array contains three rows and four columns, so it's said to be a 3-by-4 array. In general, an array with *m* rows and *n* columns is called an **m-by-n array**.

Every element in array a is identified by a name of the form `a[i][j]`, where a is the array name, and `i` and `j` are the subscripts that uniquely identify each element. The element names in row 0 all have the first subscript `0`. The element names in column 3 all have the second subscript `3`. Referencing a two-dimensional array element as `a[x, y]` instead of `a[x][y]` is a logic error. C treats `a[x, y]` as `a[y]`, so this programmer error is not a *syntax* error. The comma in this context is a **comma operator** which guarantees that a list of expressions evaluates from left to right. The value of a comma-separated list of expressions is the value of the rightmost expression in the list.

ERR

6.11.2 Initializing a Double-Subscripted Array

You can initialize a multidimensional array when you define it. For example, you can define and initialize the two-dimensional array `int b[2][2]` with:

```
int b[2][2] = {{1, 2}, {3, 4}};
```

The values in the initializer list are grouped by row in braces. The values in the first set of braces initialize row 0 and the values in the second set of braces initialize row 1. So, the values `1` and `2` initialize elements `b[0][0]` and `b[0][1]`, respectively, and the values `3` and `4` initialize elements `b[1][0]` and `b[1][1]`, respectively. If there are not enough initializers for a given row, that row's remaining elements are initialized to `0`. So the definition:

```
int b[2][2] = {{1}, {3, 4}};
```

would initialize `b[0][0]` to `1`, `b[0][1]` to `0`, `b[1][0]` to `3` and `b[1][1]` to `4`. Figure 6.16 demonstrates defining and initializing two-dimensional arrays.

```
1  // fig06_16.c
2  // Initializing multidimensional arrays.
3  #include <stdio.h>
4
5  void printArray(int a[][3]); // function prototype
6
7  // function main begins program execution
8  int main(void) {
9     int array1[2][3] = {{1, 2, 3}, {4, 5, 6}};
10    puts("Values in array1 by row are:");
11    printArray(array1);
12
13    int array2[2][3] = {{1, 2, 3}, {4, 5}};
14    puts("Values in array2 by row are:");
15    printArray(array2);
16
17    int array3[2][3] = {{1, 2}, {4}};
18    puts("Values in array3 by row are:");
19    printArray(array3);
20 }
```

Fig. 6.16 | Initializing multidimensional arrays. (Part 1 of 2.)

```
21
22  // function to output array with two rows and three columns
23  void printArray(int a[][3]) {
24     // loop through rows
25     for (size_t i = 0; i <= 1; ++i) {
26        // output column values
27        for (size_t j = 0; j <= 2; ++j) {
28           printf("%d ", a[i][j]);
29        }
30
31        printf("\n"); // start new line of output
32     }
33  }
```

```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```

Fig. 6.16 | Initializing multidimensional arrays. (Part 2 of 2.)

array1 Definition

The program defines three arrays of two rows and three columns. The definition of `array1` (line 9) provides six initializers in two sublists. The first sublist initializes *row 0* to the values 1, 2 and 3, and the second sublist initializes *row 1* to the values 4, 5 and 6.

array2 Definition

The definition of `array2` (line 13) provides five initializers in two sublists, initializing *row 0* to 1, 2 and 3, and *row 1* to 4, 5 and 0. Any elements that do not have an explicit initializer are initialized to zero automatically, so `array2[1][2]` is initialized to 0.

array3 Definition

The definition of `array3` (line 17) provides three initializers in two sublists. The first row's sublist explicitly initializes the row's first two elements to 1 and 2 and implicitly initializes the third element to 0. The second row's sublist explicitly initializes the first element to 4 and implicitly initializes the last two elements to 0.

printArray Function

The program calls `printArray` (lines 23–33) to output each array's elements. The function definition specifies the array parameter as `int a[][3]`. In a one-dimensional array parameter, the array brackets are empty. The first subscript of a multidimensional array is not required, but all subsequent subscripts are required. The compiler

uses these subscripts to determine the locations in memory of a multidimensional array's elements. All array elements are stored consecutively in memory regardless of the number of subscripts. In a two-dimensional array, the first row is stored in memory, followed by the second row.

Providing the subscript values in a parameter declaration enables the compiler to tell the function how to locate an array element. In a two-dimensional array, each row is basically a one-dimensional array. To locate an element in a particular row, the compiler must know how many elements are in each row so that it can skip the proper number of memory locations when accessing the array. So, when accessing `a[1][2]` in our example, the compiler knows to skip the three elements of the first row to get to the second row (row 1). Then, the compiler accesses element 2 of that row.

6.11.3 Setting the Elements in One Row

Many common array manipulations use `for` iteration statements. For example, the following statement sets all the elements in row 2 of the 3-by-4 `int` array `a` to zero:

```
for (int column = 0; column <= 3; ++column) {
   a[2][column] = 0;
}
```

We specified row 2, so the first subscript is always `2`. The `loop` varies only the column subscript. The preceding `for` statement is equivalent to the assignment statements:

```
a[2][0] = 0;
a[2][1] = 0;
a[2][2] = 0;
a[2][3] = 0;
```

6.11.4 Totaling the Elements in a Two-Dimensional Array

The following nested `for` statement totals the elements in the 3-by-4 `int` array `a`:

```
int total = 0;

for (int row = 0; row <= 2; ++row) {
   for (int column = 0; column <= 3; ++column) {
      total += a[row][column];
   }
}
```

The `for` statement totals the elements one row at a time. The outer `for` statement begins by setting the `row` subscript to `0` so that row's elements may be totaled by the inner `for` statement. The outer `for` statement then increments `row` to `1`, so that row's elements can be totaled. Then, the outer `for` statement increments `row` to `2`, so that row's elements can be totaled. When the nested `for` statement terminates, `total` contains the sum of all the elements in the array `a`.

6.11.5 Two-Dimensional Array Manipulations

Figure 6.17 uses `for` statements to perform several common array manipulations on a 3-by-4 array named `studentGrades`. Each row represents a student, and each column

represents a grade on one of the four exams the students took during the semester. The array manipulations are performed by four functions:

- Function `minimum` (lines 38–52) finds the lowest grade of any student for the semester.
- Function `maximum` (lines 55–69) finds the highest grade of any student for the semester.
- Function `average` (lines 72–81) calculates a particular student's semester average.
- Function `printArray` (lines 84–98) displays the two-dimensional array in a neat, tabular format.

```
1  // fig06_17.c
2  // Two-dimensional array manipulations.
3  #include <stdio.h>
4  #define STUDENTS 3
5  #define EXAMS 4
6
7  // function prototypes
8  int minimum(const int grades[][EXAMS], size_t pupils, size_t tests);
9  int maximum(const int grades[][EXAMS], size_t pupils, size_t tests);
10 double average(const int setOfGrades[], size_t tests);
11 void printArray(const int grades[][EXAMS], size_t pupils, size_t tests);
12
13 // function main begins program execution
14 int main(void) {
15    // initialize student grades for three students (rows)
16    int studentGrades[STUDENTS][EXAMS] =
17       {{77, 68, 86, 73},
18        {96, 87, 89, 78},
19        {70, 90, 86, 81}};
20
21    // output array studentGrades
22    puts("The array is:");
23    printArray(studentGrades, STUDENTS, EXAMS);
24
25    // determine smallest and largest grade values
26    printf("\n\nLowest grade: %d\nHighest grade: %d\n",
27       minimum(studentGrades, STUDENTS, EXAMS),
28       maximum(studentGrades, STUDENTS, EXAMS));
29
30    // calculate average grade for each student
31    for (size_t student = 0; student < STUDENTS; ++student) {
32       printf("The average grade for student %zu is %.2f\n",
33          student, average(studentGrades[student], EXAMS));
34    }
35 }
```

Fig. 6.17 | Two-dimensional array manipulations. (Part 1 of 3.)

```
36
37 // Find the minimum grade
38 int minimum(const int grades[][EXAMS], size_t pupils, size_t tests) {
39    int lowGrade = 100; // initialize to highest possible grade
40
41    // loop through rows of grades
42    for (size_t row = 0; row < pupils; ++row) {
43       // loop through columns of grades
44       for (size_t column = 0; column < tests; ++column) {
45          if (grades[row][column] < lowGrade) {
46             lowGrade = grades[row][column];
47          }
48       }
49    }
50
51    return lowGrade; // return minimum grade
52 }
53
54 // Find the maximum grade
55 int maximum(const int grades[][EXAMS], size_t pupils, size_t tests) {
56    int highGrade = 0; // initialize to lowest possible grade
57
58    // loop through rows of grades
59    for (size_t row = 0; row < pupils; ++row) {
60       // loop through columns of grades
61       for (size_t column = 0; column < tests; ++column) {
62          if (grades[row][column] > highGrade) {
63             highGrade = grades[row][column];
64          }
65       }
66    }
67
68    return highGrade; // return maximum grade
69 }
70
71 // Determine the average grade for a particular student
72 double average(const int setOfGrades[], size_t tests) {
73    int total = 0; // sum of test grades
74
75    // total all grades for one student
76    for (size_t test = 0; test < tests; ++test) {
77       total += setOfGrades[test];
78    }
79
80    return (double) total / tests; // average
81 }
82
83 // Print the array
84 void printArray(const int grades[][EXAMS], size_t pupils, size_t tests) {
85    // output column heads
86    printf("%s", "                [0]  [1]  [2]  [3]");
87
```

Fig. 6.17 | Two-dimensional array manipulations. (Part 2 of 3.)

```
88     // output grades in tabular format
89     for (size_t row = 0; row < pupils; ++row) {
90        // output label for row
91        printf("\nstudentGrades[%zu] ", row);
92
93        // output grades for one student
94        for (size_t column = 0; column < tests; ++column) {
95           printf("%-5d", grades[row][column]);
96        }
97     }
98  }
```

```
The array is:
                  [0]  [1]  [2]  [3]
studentGrades[0] 77   68   86   73
studentGrades[1] 96   87   89   78
studentGrades[2] 70   90   86   81

Lowest grade: 68
Highest grade: 96
The average grade for student 0 is 76.00
The average grade for student 1 is 87.50
The average grade for student 2 is 81.75
```

Fig. 6.17 | Two-dimensional array manipulations. (Part 3 of 3.)

Nested Loops in Functions `minimum`, `maximum` and `printArray`

Functions `minimum`, `maximum` and `printArray` each receive three arguments—the `studentGrades` array (called `grades` in each function), the number of students (rows in the array) and the number of exams (columns in the array). Each function loops through array `grades` using nested `for` statements. The following nested `for` statement is from the function `minimum` definition:

```
// loop through rows of grades
for (size_t row = 0; row < pupils; ++row) {
   // loop through columns of grades
   for (size_t column = 0; column < tests; ++column) {
      if (grades[row][column] < lowGrade) {
         lowGrade = grades[row][column];
      }
   }
}
```

The outer `for` statement begins by setting `row` to `0` so that row's elements (i.e., the first student's grades) can be compared to variable `lowGrade` in the inner `for` statement. The inner `for` statement loops through a particular row's four grades and compares each grade to `lowGrade`. If a grade is less than `lowGrade`, the nested `if` statement sets `lowGrade` to that grade. The outer `for` statement then increments `row` to `1`, and that row's elements are compared to `lowGrade`. The outer `for` statement then increments `row` to `2`, and that row's elements are compared to `lowGrade`. When the nested

statement completes execution, `lowGrade` contains the smallest grade in the two-dimensional array. Function `maximum` works similarly to function `minimum`.

Function average

Function `average` (lines 72–81) takes two arguments—a one-dimensional array of test results for a particular student (`setOfGrades`) and the number of test results in the array. When line 33 calls `average`, the first argument—`studentGrades[student]`—passes the address of one row of the two-dimensional array. The argument `studentGrades[1]` is the starting address of row 1 of the array. Remember that a two-dimensional array is basically an array of one-dimensional arrays, and the name of a one-dimensional array is the address of that array in memory. Function `average` calculates the sum of the array elements, divides the total by the number of test results and returns the floating-point result.

Self Check

1 *(What Does This Code Do?)* What does the following nested `for` statement do?

```
product = 1;

for (row = 0; row <= 2; ++row) {
   for (column = 0; column <= 3; ++column) {
      product *= m[row][column];
   }
}
```

Answer: It calculates the `product` of all the element values in `3`-by-`4` double array `m`.

2 *(What Does This Code Do?)* What does the following nested `for` statement do?

```
// loop through rows of grades
for (i = 0; i < pupils; ++i) {
   // loop through columns of grades
   for (j = 0; j < tests; ++j) {
      if (grades[i][j] < lowGrade) {
         lowGrade = grades[i][j];
      }
   }
}
```

Answer: It loops through a two-dimensional array `grades` with `pupils` rows and `tests` columns attempting to find the minimum grade in the array. Assuming the grades would be zero through `100`, `lowGrade` would need to be initialized to a value of `100` or greater.

6.12 Variable-Length Arrays

For each array you've defined so far, you've specified its size at compilation time. But what if you cannot determine an array's size until execution time? In the past, to handle this, you had to use dynamic memory allocation (introduced in Chapter 12, Data Structures). For cases in which an array's size is not known at compilation time, C

has **variable-length arrays** (VLAs)—arrays whose lengths are determined by expressions evaluated at execution time.[2] The program of Fig. 6.18 declares and prints several VLAs.

```
1  // fig06_18.c
2  // Using variable-length arrays in C99
3  #include <stdio.h>
4
5  // function prototypes
6  void print1DArray(size_t size, int array[size]);
7  void print2DArray(size_t row, size_t col, int array[row][col]);
8
9  int main(void) {
10    printf("%s", "Enter size of a one-dimensional array: ");
11    int arraySize = 0; // size of 1-D array
12    scanf("%d", &arraySize);
13
14    int array[arraySize]; // declare 1-D variable-length array
15
16    printf("%s", "Enter number of rows and columns in a 2-D array: ");
17    int row1 = 0; // number of rows in a 2-D array
18    int col1 = 0; // number of columns in a 2-D array
19    scanf("%d %d", &row1, &col1);
20
21    int array2D1[row1][col1]; // declare 2-D variable-length array
22
23    printf("%s",
24       "Enter number of rows and columns in another 2-D array: ");
25    int row2 = 0; // number of rows in a 2-D array
26    int col2 = 0; // number of columns in a 2-D array
27    scanf("%d %d", &row2, &col2);
28
29    int array2D2[row2][col2]; // declare 2-D variable-length array
30
31    // test sizeof operator on VLA
32    printf("\nsizeof(array) yields array size of %zu bytes\n",
33       sizeof(array));
34
35    // assign elements of 1-D VLA
36    for (size_t i = 0; i < arraySize; ++i) {
37       array[i] = i * i;
38    }
39
40    // assign elements of first 2-D VLA
41    for (size_t i = 0; i < row1; ++i) {
42       for (size_t j = 0; j < col1; ++j) {
43          array2D1[i][j] = i + j;
44       }
45    }
46
```

Fig. 6.18 | Using variable-length arrays in C99. (Part 1 of 3.)

2. This feature is not supported in Microsoft Visual C++.

```
47     // assign elements of second 2-D VLA
48     for (size_t i = 0; i < row2; ++i) {
49        for (size_t j = 0; j < col2; ++j) {
50           array2D2[i][j] = i + j;
51        }
52     }
53
54     puts("\nOne-dimensional array:");
55     print1DArray(arraySize, array); // pass 1-D VLA to function
56
57     puts("\nFirst two-dimensional array:");
58     print2DArray(row1, col1, array2D1); // pass 2-D VLA to function
59
60     puts("\nSecond two-dimensional array:");
61     print2DArray(row2, col2, array2D2); // pass other 2-D VLA to function
62  }
63
64  void print1DArray(size_t size, int array[size]) {
65     // output contents of array
66     for (size_t i = 0; i < size; i++) {
67        printf("array[%zu] = %d\n", i, array[i]);
68     }
69  }
70
71  void print2DArray(size_t row, size_t col, int array[row][col]) {
72     // output contents of array
73     for (size_t i = 0; i < row; ++i) {
74        for (size_t j = 0; j < col; ++j) {
75           printf("%5d", array[i][j]);
76        }
77
78        puts("");
79     }
80  }
```

```
Enter size of a one-dimensional array: 6
Enter number of rows and columns in a 2-D array: 2 5
Enter number of rows and columns in another 2-D array: 4 3

sizeof(array) yields array size of 24 bytes

One-dimensional array:
array[0] = 0
array[1] = 1
array[2] = 4
array[3] = 9
array[4] = 16
array[5] = 25

First two-dimensional array:
    0    1    2    3    4
    1    2    3    4    5
```

Fig. 6.18 | Using variable-length arrays in C99. (Part 2 of 3.)

```
Second two-dimensional array:
    0    1    2
    1    2    3
    2    3    4
    3    4    5
```

Fig. 6.18 | Using variable-length arrays in C99. (Part 3 of 3.)

Creating the VLAs

Lines 10–29 prompt the user for the desired sizes for a one-dimensional array and two two-dimensional arrays and use the input values in lines 14, 21 and 29 to create VLAs. These lines are valid as long as the variables representing the array sizes are integers.

`sizeof` Operator with VLAs

After creating the arrays, we use the `sizeof` operator in lines 32–33 to check our one-dimensional VLA's length. Operator `sizeof` is normally a compile-time operation, but it operates at runtime when applied to a VLA. The output window shows that the `sizeof` operator returns a size of 24 bytes—four times the number we entered because the size of an `int` on our machine is 4 bytes.

Assigning Values to VLA Elements

Next, we assign values to our VLAs' elements (lines 36–52). We use the loop-continuation condition `i < arraySize` when filling the one-dimensional array. As with fixed-length arrays, there's no protection against stepping outside the array bounds.

Function `print1DArray`

Lines 64–69 define function `print1DArray` that displays its one-dimensional VLA argument. VLA function parameters have the same syntax as regular array parameters. We use the parameter `size` in parameter `array`'s declaration, but it's purely documentation for the programmer.

Function `print2DArray`

Function `print2DArray` (lines 71–80) displays a two-dimensional VLA. Recall that you must specify a size for all but the first subscript in a multidimensional array parameter. The same restriction holds true for VLAs, except that the sizes can be specified by variables. The initial value of `col` passed to the function determines where each row begins in memory, just as with a fixed-size array.

Self Check

1 *(True/False)* Unlike with fixed-length arrays, VLAs offer protection against stepping outside the array bounds.

Answer: *False*. Actually, as with fixed-length arrays, there is no protection against stepping outside the array bounds.

2 *(True/False)* `sizeof` is a compile-time-only operation.
Answer: *False*. Actually, generally `sizeof` is a compile-time operation, but when applied to a VLA, `sizeof` operates at runtime.

6.13 Secure C Programming

SEC

Bounds Checking for Array Subscripts

It's important to ensure that every subscript used to access an array element is within the array's bounds. A one-dimensional array's subscripts must be greater than or equal to 0 and less than the number of elements. A two-dimensional array's row and column subscripts must be greater than or equal to 0 and less than the numbers of rows and columns, respectively. This also applies to arrays with additional dimensions.

Allowing programs to read from or write to array elements outside an array's bounds are common security flaws. Reading from out-of-bounds array elements can cause a program to crash or even appear to execute correctly while using bad data. Writing to an out-of-bounds element (known as a buffer overflow) can corrupt a program's data in memory, crash a program and even allow attackers to exploit the system and execute their own code.

C provides no automatic bounds checking for arrays. You must ensure that array subscripts are always greater than or equal to 0 and less than the array's number elements. For additional techniques that help you prevent such problems, see CERT guideline ARR30-C at `https://wiki.sei.cmu.edu/confluence`.

`scanf_s`

Bounds checking is also important in string processing. When reading a string into a `char` array, `scanf` does not automatically prevent buffer overflows. If the number of characters input is greater than or equal to the array's length, `scanf` will write characters—including the string's terminating null character (`'\0'`)—beyond the end of the array. This might overwrite other variables' values in memory. In addition, if the program writes to those other variables, it might overwrite the string's `'\0'`.

A function determines where a string ends by looking for its terminating `'\0'` character. For example, recall that function `printf` outputs a string by reading characters from the beginning of the string in memory and continuing until it encounters the string's `'\0'`. If the `'\0'` is missing, `printf` continues reading from memory (and printing) until it encounters some later `'\0'` in memory. This can lead to strange results or cause a program to crash.

The C11 standard's optional Annex K provides more secure versions of many string-processing and input/output functions. When reading a string into a character array, function `scanf_s` performs checks to ensure that it does not write beyond the end of the array. Assuming that `myString` is a 20-character array, the statement

```
scanf_s("%19s", myString, 20);
```

reads a string into `myString`. Function `scanf_s` requires two arguments for each `%s` in the format string:

- a character array in which to place the input string and
- the array's number of elements.

The function uses the number of elements to prevent buffer overflows. For example, it's possible to supply a field width for `%s` that's too long for the underlying character array, or to simply omit the field width entirely. With `scanf_s`, if the number of characters input plus the terminating null character is larger than the number of array elements specified, the `%s` conversion fails. For the preceding statement, which contains only one conversion specification, `scanf_s` would return `0`, indicating no conversions were performed. The array `myString` would be unaltered. We discuss additional Annex K functions in later Secure C Programming sections.

Not all compilers support the C11 standard's Annex K functions. For programs that must compile on multiple platforms and compilers, you might have to edit your code to use the versions of `scanf_s` or `scanf` available on each platform. Your compiler might also require a specific setting to enable you to use the Annex K functions.

Don't Use Strings Read from the User as Format-Control Strings

You might have noticed that throughout this book, we do not use single-argument `printf` statements. Instead, we use one of the following forms:

- When we need to output a `'\n'` after the string, we use function `puts` (which automatically outputs a `'\n'` after its single string argument), as in

```
puts("Welcome to C!");
```

- When we need the cursor to remain on the same line as the string, we use function `printf`, as in

```
printf("%s", "Enter first integer: ");
```

Because we were displaying string literals, we certainly could have used the one-argument form of `printf`, as in

```
printf("Welcome to C!\n");
printf("Enter first integer: ");
```

When `printf` evaluates the format-control string in its first (and possibly only) argument, it performs tasks based on the conversion specification(s) in that string. If the format-control string were obtained from the user, an attacker could supply malicious conversion specifications that would be "executed" by the formatted output function. Now that you know how to read strings into character arrays, it's important to note that you should never use as a `printf`'s format-control string a character array that might contain user input. For more information, see CERT guideline FIO30-C at `https://wiki.sei.cmu.edu/confluence`.

Self Check

1 *(Multiple Choice)* Which of the following statements a), b) or c) is *false*?

a) It's important to ensure that every index you use to access an array element is within the array's bounds. A one-dimensional array's indexes must be greater

than or equal to 0 and less than the number of array elements. A two-dimensional array's row and column indexes must be greater than or equal to 0 and less than the numbers of rows and columns, respectively.

b) C provides no automatic bounds checking for arrays, so you must provide your own. Allowing programs to read from or write to array elements outside the bounds of arrays are common security flaws.

c) Reading from out-of-bounds array elements can cause a program to crash or even appear to execute correctly while using bad data.

d) All of the above statements are *true*.

Answer: d.

2 ***(True/False)*** When `printf` evaluates the format-control string in its first (and possibly its only) argument, the function performs tasks based on the conversion specifier(s) in that string. If the format-control string were obtained from the user, an attacker could supply malicious conversion specifiers that would be "executed" by the formatted output function. You should never use as a `printf`'s format-control string a character array that might contain user input.

Answer: *True*.

Summary

Section 6.1 Introduction

- **Arrays** (p. 298) are data structures consisting of related data items of the same type.
- Arrays are "static" entities in that they remain the same size throughout program execution.

Section 6.2 Arrays

- An array is a **contiguous group of memory locations** related by the fact that they all have the same name and the same type.
- To refer to a particular **location** or **element** (p. 298), specify the array's name and the **position number** (p. 299) of the element.
- The first element in every array is at location 0.
- The position number contained within square brackets is more formally called a **subscript** (p. 299) or **index**. A subscript must be an integer or an integer expression.
- The **brackets** that enclose an array subscript are an operator with the highest level of precedence.

Section 6.3 Defining Arrays

- You specify the array's element type and **number of elements** so that the computer may reserve the appropriate amount of memory.
- An **array of type `char`** can be used to store a **character string**.

Section 6.4 Array Examples

- Type **`size_t`** represents an **unsigned integral type**. This type is recommended for any variable that represents an array's size or an array's subscripts. The **header `<stddef.h>`** defines `size_t` and is often included by other headers (such as `<stdio.h>`).

- An array's elements can be initialized when the array is defined by following the definition with an equals sign and braces, `{}`, containing a comma-separated **list of initializers** (p. 302). If there are fewer initializers than array elements, the remaining elements are initialized to zero.
- The statement `int n[10] = {0};` explicitly initializes the first element to zero and initializes the remaining nine elements to zero because there are fewer initializers than array elements.
- If the array size is omitted from a definition with an initializer list, the compiler determines the array's number of elements from the number of initializers.
- The **`#define` preprocessor directive** can define a **symbolic constant**—an identifier that the preprocessor replaces with replacement text before the program is compiled. When a program is preprocessed, all occurrences of the symbolic constant are replaced with the replacement text (p. 303). Using symbolic constants to specify array sizes makes programs easier to read and more **modifiable**.
- C has **no array bounds checking** to prevent a program from referring to an element that does not exist. Thus, an executing program can "walk off" the end of an array without warning. You should ensure that all array references remain within the bounds of the array.

Section 6.5 Using Character Arrays to Store and Manipulate Strings

- A string literal such as `"hello"` is really an array of individual characters in C.
- A **character array** can be initialized using a **string literal**. In this case, the array's size is determined by the compiler based on the string's length.
- Every string contains a special **string-termination character** called the **null character** (p. 309). The character constant representing the null character is `'\0'`.
- A character array representing a string should always be defined large enough to hold the number of characters in the string and the terminating null character.
- Character arrays also can be initialized with individual character constants in an initializer list.
- Because a string is really an array of characters, we can access individual characters in a string directly using array subscript notation.
- You can input a string directly into a character array from the keyboard using `scanf` and the **conversion specification `%s`**. The character array's name is passed to `scanf` without the preceding `&` used with non-array variables.
- Function `scanf` reads characters from the keyboard until the first whitespace character is encountered—it does not check the array size. Thus, `scanf` can write beyond the end of an array. For this reason, when reading a string into a `char` array with `scanf`, you should always use a field width that's one less than the `char` array's size (e.g., `"%19s"` for a 20-`char` array).
- A character array representing a string can be output with `printf` and the `%s` conversion specification. The characters of the string are printed until a terminating null character is encountered.

Section 6.6 Static Local Arrays and Automatic Local Arrays

- We can apply `static` to a local array definition so the function does not create and initialize the array in each call and destroy it each time the function exits. This reduces program execution time, particularly for programs with frequently called functions that contain large arrays.
- Arrays that are `static` are automatically initialized once at program startup. If you do not explicitly initialize a `static` array, that array's elements are initialized to zero by the compiler.

Section 6.7 Passing Arrays to Functions

- To pass an array argument to a function, specify the array name without any brackets.
- Unlike `char` arrays that contain strings, other array types do not have a special terminator. For this reason, the array's size is passed to a function, so the function can process the proper number of elements.
- **C automatically passes arrays to functions by reference**—the called functions can modify the element values in the callers' original arrays. An array name evaluates to the address of the array's first element. Because the starting address of the array is passed, the called function knows precisely where the array is stored and can modify the original array in the caller.
- Although entire arrays are passed by reference, **individual array elements are passed by value** exactly as simple variables are.
- Single pieces of data (such as individual `int`s, `float`s and `char`s) are called **scalars** (p. 315).
- To pass an array element to a function, use the subscripted name of the array element.
- For a function to receive an array through a function call, the function's parameter list must specify that an array will be received. The array's size is not required between the array brackets. If it's included, the compiler checks that it's greater than zero, then ignores it.
- If an array parameter is preceded by the **`const` qualifier** (p. 317), any attempt to modify an element in the function body results in a compilation error.

Section 6.8 Sorting Arrays

- **Sorting** data—that is, placing the data into ascending or descending order—is one of the most important computing applications.
- One sorting technique is called the **bubble sort** (p. 319) or the **sinking sort**, because the smaller values gradually "bubble" their way upward to the top of the array like air bubbles rising in water, while the larger values sink to the bottom of the array. The technique is to make several passes through the array. On each pass, successive pairs of elements are compared. If a pair is in increasing order (or the values are identical), we leave the values as they are. If a pair is in decreasing order, their values are swapped in the array.
- Bubble sort may move a large value down the array many positions on a single pass but may move a small value up only one position.
- The chief virtue of the bubble sort is that it's easy to program. However, it runs slowly. This becomes apparent when sorting large arrays.

Section 6.9 Intro to Data Science Case Study: Survey Data Analysis

- The **mean** is the arithmetic average of a set of values.
- The **median** is the "middle value" in a sorted set of values.
- The **mode** is the value that occurs most frequently in a set of values.

Section 6.10 Searching Arrays

- The process of finding a particular array element is called **searching** (p. 326).
- The **linear search** compares each array element with a search key (p. 326). The array is not in any particular order, so it's just as likely that the value will be found in the first element as in the last. On average, therefore, the search key will be compared with half the array's elements.
- The linear search algorithm (p. 326) works well for small or unsorted arrays. For sorted arrays, the high-speed binary search algorithm can be used.

- The **binary search** algorithm (p. 326) eliminates from consideration one-half of a sorted array's elements after each comparison. The algorithm locates the middle array element and compares it to the search key. If they're equal, the search key is found, and that element's subscript is returned. If they're not equal, the problem is reduced to searching one-half of the array. If the search key is less than the middle array element, the array's first half is searched; otherwise, the second half is searched. If the search key is not found in the specified subarray, the algorithm is repeated on one-quarter of the original array. The search continues until the search key is equal to a subarray's middle element, or until the subarray consists of one element that's not equal to the search key (i.e., the search key is not found).
- When using a binary search, the maximum number of comparisons required for any array can be determined by finding the first power of 2 greater than the number of array elements.

Section 6.11 Multidimensional Arrays

- A common use of **multidimensional arrays** (p. 332) is to represent **tables** of values consisting of **rows** and **columns**. To identify a particular table element, we specify two subscripts. By convention, the first identifies the element's row, and the second identifies its column.
- Arrays that require two subscripts to identify an element are called **two-dimensional arrays** (p. 332). Multidimensional arrays can have more than two subscripts.
- A multidimensional array can be initialized when it's defined. The values in a two-dimensional array's initializer list are grouped by row in braces. If there are not enough initializers for a given row, its remaining elements are initialized to `0`.
- The first subscript of a multidimensional array parameter declaration is not required, but all subsequent subscripts are. The compiler uses these sizes to determine the locations in memory of elements in multidimensional arrays. All array elements are stored consecutively in memory, regardless of the number of subscripts. In a two-dimensional array, the first row is stored in memory, followed by the second row, and so on.
- Providing the subscript values in a parameter declaration enables the compiler to tell the function how to locate an array element. In a two-dimensional array, each row is basically a one-dimensional array. To locate an element in a particular row, the compiler must know how many elements are in each row, so it can skip the proper number of memory locations when accessing elements of a given row.

Section 6.12 Variable-Length Arrays

- A **variable-length array** (p. 340) is an array for which the size is defined by an expression evaluated at execution time.
- When applied to a variable-length array, **`sizeof`** operates at runtime.
- Variable-length arrays are optional in C—they may not be supported by your compiler.

Self-Review Exercises

6.1 Answer each of the following:

a) Lists and tables of values are stored in _____.
b) The number used to refer to a particular array element is called its _____.
c) A(n) _____ should be used to specify the size of an array because it makes the program more modifiable.
d) Placing the elements of an array in order is called _____ the array.
e) Determining whether an array contains a key value is called _____ the array.
f) An array that uses two subscripts is referred to as a(n) _____ array.

6.2 State whether the following are *true* or *false*. If the answer is *false*, explain why.
a) An array can store many different types of values.
b) An array subscript can be of data type `double`.
c) If there are fewer initializers in an initializer list than there are array elements, the remaining elements are initialized with the initializer list's last value.
d) It's an error if an initializer list contains more initializers than there are array elements.
e) An individual array element that's passed to a function as an argument of the form `a[i]` and modified in the called function will contain the modified value in the calling function.

6.3 Follow the instructions below regarding an array called `fractions`.
a) Define a symbolic constant `SIZE` with the replacement text 10.
b) Define a `double` array with `SIZE` elements and initialize the elements to `0`.
c) Refer to array element 4.
d) Assign the value `1.667` to array element nine.
e) Assign the value `3.333` to the seventh element of the array.
f) Print array elements 6 and 9 with two digits of precision to the right of the decimal point, and show the output that's displayed on the screen.
g) Print all the elements of an array using a `for` iteration statement. Use the variable `x` as the loop's control variable. Show the output.

6.4 Write statements to accomplish the following:
a) Define `table` to be an integer array and to have 3 rows and 3 columns. Assume the symbolic constant `SIZE` has been defined to be 3.
b) How many elements does the array `table` contain? Print the total number of elements.
c) Use a `for` iteration statement to initialize each element of `table` to the sum of its subscripts. Use variables `x` and `y` as control variables.
d) Print the values of each element of array `table`. Assume the array was initialized with the definition:

```
int table[SIZE][SIZE] = {{1, 8}, {2, 4, 6}, {5}};
```

6.5 Find the error in each of the following program segments and correct the error.
a)
```
#define SIZE 100;
```
b)
```
SIZE = 10;
```
c)
```
int b[10] = {0};
int i;
for (size_t i = 0; i <= 10; ++i) {
   b[i] = 1;
}
```
d)
```
#include <stdio.h>;
```
e)
```
int a[2][2] = {{1, 2}, {3, 4}};
a[1, 1] = 5;
```
f)
```
#define VALUE = 120
```

Answers to Self-Review Exercises

6.1 a) arrays. b) subscript (or index). c) symbolic constant. d) sorting. e) searching. f) two-dimensional.

6.2 a) *False*. An array can store only values of the same type.
b) *False*. An array subscript must be an integer or an integer expression.
c) *False*. C automatically initializes the remaining elements to zero.
d) *True*.
e) *False*. Individual elements of an array are passed by value. If the entire array is passed to a function, any modifications will be reflected in the original array.

6.3 a)
```
#define SIZE 10
```
b)
```
double fractions[SIZE] = {0.0};
```
c)
```
fractions[4]
```
d)
```
fractions[9] = 1.667;
```
e)
```
fractions[6] = 3.333;
```
f)
```
printf("%.2f %.2f\n", fractions[6], fractions[9]);
```
Output: `3.33 1.67`.
g)
```
for (size_t x = 0; x < SIZE; ++x) {
   printf("fractions[%zu] = %f\n", x, fractions[x]);
}
```

Output:
```
fractions[0] = 0.000000
fractions[1] = 0.000000
fractions[2] = 0.000000
fractions[3] = 0.000000
fractions[4] = 0.000000
fractions[5] = 0.000000
fractions[6] = 3.333000
fractions[7] = 0.000000
fractions[8] = 0.000000
fractions[9] = 1.667000
```

6.4 a)
```
int table[SIZE][SIZE];
```
b) Nine elements. `printf("%d\n", SIZE * SIZE);`
c)
```
for (size_t x = 0; x < SIZE; ++x) {
   for (size_t y = 0; y < SIZE; ++y) {
      table[x][y] = x + y;
   }
}
```
d)
```
for (size_t x = 0; x < SIZE; ++x) {
   for (size_t y = 0; y < SIZE; ++y) {
      printf("table[%d][%d] = %d\n", x, y, table[x][y]);
   }
}
```

Output:
```
table[0][0] = 1
```

```
table[0][1] = 8
table[0][2] = 0
table[1][0] = 2
table[1][1] = 4
table[1][2] = 6
table[2][0] = 5
table[2][1] = 0
table[2][2] = 0
```

6.5 a) Error: Semicolon at the end of the `#define` preprocessor directive.
Correction: Eliminate semicolon.

b) Error: Assigning a value to a symbolic constant using an assignment statement.
Correction: Assign a value to the symbolic constant in a `#define` preprocessor directive without using the assignment operator, as in `#define SIZE 10`.

c) Error: Referencing an array element outside the bounds of the array (`b[10]`).
Correction: Change the control variable's final value to `9` or change `<=` to `<`.

d) Error: Semicolon at the end of the `#include` preprocessor directive.
Correction: Eliminate semicolon.

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

f) Error: A symbolic constant's value is not defined using `=`.
Correction: Change the preprocessor directive to `#define VALUE 120`.

Exercises

6.6 Fill in the blanks in each of the following:

a) Arrays are _____ entities as they remain the same size during program execution.

b) An array is a _____ group of memory locations that share the same _____.

c) The first element in every array is the _____ element, for example, the first element of an array is referred to as _____.

d) Array elements are referred by _____ followed by the _____ of the element.

e) Arrays are passed to functions by_____.

f) C has no array _____, which means a program can accidentally store data passing the bound of the array.

g) The _____ operator cannot be used directly with arrays for copying data from one array to another.

h) Every string contains a special string-termination character called the _____ character, represented by the character constant _____.

i) A 5-by-7 array contains _____ rows, _____ columns, and _____ elements.

j) The name of the element in the second row and third column of an array would be _____ .

6.7 State which of the following are *true* and which are *false*. If *false*, explain why.

a) If there are fewer initializers in an initializer list than the number of elements in the array, the remaining elements will contain garbage values.

b) A static local variable in a function definition exists for the duration of the program.

c) The type size_t represents an unsigned integral type.

d) If the array size is omitted from a definition with an initializer list, the number of elements in the array will not be defined.

e) To calculate the amount of memory used by an array, multiply the number of elements by the size in bytes each element occupies.

f) When individual array elements are passed to a function, they are passed by reference, and their modified value is retained in the calling function.

6.8 Write statements to accomplish each of the following:

a) Multiply the value of element 4 of an integer array `n` by 3 and display it.

b) Write a loop that adds all the elements of the array n[**10**] and stores the result in `total`.

c) Initialize each of the 9 elements of a two-dimensional integer array `m[3][3]` to 3, using loops.

d) Find the largest and smallest element of a two-dimensional array `sales[4][5]`.

e) Copy a 100-element array into a 200-element array, starting from the 100th position of the larger array.

f) Determine and store the sum and difference of the values contained in two, 100-element double arrays `d1` and `d2`, into double arrays `sum` and `difference`.

6.9 Consider a 5-by-20 integer array `grades`:

a) Write a declaration for `grades`.

b) How many rows does the array have?

c) How many columns does the array have?

d) How many elements does the array have?

e) Write the names of all elements in the first column of the array.

f) Write the name of the element in the third row and second column of the array.

g) Write a single statement to assign the value 100 to the element in the first row and second column.

h) Write a nested loop to get all the elements from the keyboard.

i) Write a nested `for` statement to initialize all elements to zero.

j) Write a statement that copies the values from an array `double mathGrades[20]` into the elements of the first row of marks.

k) Write a series of statements that determine and print the highest value in first row of `grades`.

l) Write a statement to display the elements in column 2 of the array.

m) Write a statement to calculate the average of the elements in the first row.

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

6.10 ***(Sales Commissions)*** Use a one-dimensional array to solve the following problem. A company pays its salespeople on a commission basis. The salespeople receive $200 per week plus 9% of their gross sales for that week. For example, a salesperson who grosses $3000 in sales in a week receives $200 plus 9% of $3000 for a total of $470. Write a C program (using an array of counters) that determines how many 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

6.11 ***(Selection Sort)*** A *selection sort algorithm* for a one-dimensional array has the following steps:

1. The smallest value in the array is found.
2. It is swapped with the value in the first position of the array.
3. The above steps are repeated for the rest of the array starting at the second position and advancing each time.

Eventually the entire array is divided into two parts: the sub-array of items already sorted which is built up from left to right and is found at the beginning, and the sub-array of items remaining to be sorted, occupying the remainder of the array. Write a program that sorts an array of 10 integers using this algorithm.

6.12 Write loops that perform each of the following one-dimensional array operations:

a) Read the 20 elements of double array `sales` from the keyboard.
b) Add 1000 to each of the 75 elements of double array `allowance`.
c) Initialize the 50 elements of integer array `numbers` to zero.
d) Print the 10 values of integer array `GPA` in column format.

6.13 Find the error(s) in each of the following statements:

a) Assume: `int a[5];`

```
scanf("%d", a[5]);
```

b) Assume: `int a[3][3];`

```
printf("%d%d%d\n", a[0][1], a[0][2], a[0][3]);
```

c)
```
double f[3] = {1.1; 10.01; 100.001, };
```

d) Assume: `double d[3][5];`

```
d[2, 4] = 2.345;
```

6.14 *(Union of Sets)* Use one-dimensional arrays to solve the following problem. Read in two sets of numbers, each having 10 numbers. After reading all values, display all the unique elements in the collection of both sets of numbers. Use the smallest possible array to solve this problem.

6.15 *(Intersection of Sets)* Use one-dimensional arrays to solve the following problem. Read in two sets of numbers, each having 10 numbers. After reading all values, display the unique elements common to both sets of numbers.

6.16 Label the elements of 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 <= 2; ++row) {
   for (size_t column = 0; column <= 4; ++column) {
      sales[row][column] = 0;
   }
}
```

6.17 What does the following program do?

```
#include <stdio.h>
#define SIZE 6

int whatIsThis(const int b[], size_t p);
// function prototype

// function main begins program execution
int main(void)
{
   int x; // holds return value of function whatIsThis
   // initialize array a
   int a[SIZE] = {1, 2, 3, 4, 5, 6};
   x = whatIsThis(a, SIZE);
   printf("Result is %d\n", x);
}

// what does this function do?
int whatIsThis(const int b[], size_t p)
{
   // base case
   if (1 == p)
   {
      return b[0];
   }
   else { // recursion step
      return b[p - 1] * whatIsThis(b, p - 1);
   }
}
```

6.18 What does the following program do?

```
#include <stdio.h>
   #define MAX 10
   /* function prototype */
```

```
4  void functionName(const int b[], size_t startSubscript, size_t size);
5
6  /* function main begins program execution */
7  int main(void)
8  {
9
10    /* initialize p */
11    int p[MAX] = {5, 7, 2, 1, 0, 4, 3, 0, 6, 8};
12
13    puts("Answer is:");
14    functionName(p, 0, MAX);
15    puts("");
16 } /* end main */
17
18    /* What does this function do? */
19 void functionName(const int b[], size_t startSubscript, size_t size)
20 {
21    if (startSubscript < size) {
22       functionName(b, startSubscript + 1, size);
23       printf("%d", b[startSubscript] * 5);
24    } /* end if */
25 } /* end function someFunction */
```

6.19 ***(Dice Rolling)*** Write a program that simulates rolling two dice. The program should use `rand` twice to roll the first and second die, respectively, then calculate their sum. Because each die can have an integer value from 1 to 6, the sum of the values will vary from 2 to 12, with 7 being the most frequent sum and 2 and 12 the least frequent sums. The following diagram shows the 36 possible combinations of the two dice:

	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

Your program should roll the two dice 36,000 times. Use a one-dimensional array to tally the numbers of times each possible sum appears. Print the results in tabular format. Also, determine whether the totals are reasonable—for example, there are six ways to roll a 7, so approximately one-sixth of all the rolls should be 7.

6.20 ***(Rock, Paper, Scissors Game Statistics)*** Write a program, modified from **Fig. 5.7** and **Exercise 5.47**, that runs 1,000,000 games (without human intervention) and answers each of the following questions.

a) How many games are won by the player?
b) What is the average length (i.e., number of rounds) of a game?

c) What are the chances of winning at a rock, paper, scissors game with the strategy you have implemented in Exercise 5.7?
d) What are the chances of winning if this is a **best of 99** instead of **best of 3**?
e) If the player plays with random strategy to pick the shape, is the chance of winning higher than the strategy we have implemented in Exercise 5.7? [*Hint*: modify the program and rerun the 1,000,000 games with the player picking random shapes as a strategy].
f) What have we learned from this exercise?

6.21 ***(Matrix Multiplication)*** An $n \times m$ two-dimensional matrix can be multiplied by another $m \times p$ matrix to give a matrix whose elements are the sum of the products of the elements within a row from the first matrix and the associated elements of a column of the second matrix. Both matrices should either be square matrices, or the number of columns of the first matrix should equal the number of rows of the second matrix.

To calculate each element of the resultant matrix, multiply the first element of a given row from the first matrix and the first element of a given column in the second matrix, add that to the product of the second element of the same row and the second element of the same column, and keep doing so until the last elements of the row and column have been multiplied and added to the sum.

Write a program to calculate the product of 2 matrices, and store the result in a third matrix.

6.22 ***(Total Sales)*** 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:

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 reads all this sales information and summarizes 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 column representing a particular salesperson and each row 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.

6.23 ***(Turtle Graphics)*** The Logo language made the concept of *turtle graphics* famous. Imagine a mechanical turtle that walks around the room under the control of a C program. The turtle holds a pen in one of two positions, up or down. While the pen is down, the turtle traces out shapes as it moves; while the pen is up, the turtle

moves about freely without writing anything. In this problem, you'll simulate the operation of the turtle and create a computerized sketchpad as well.

Use a 50-by-50 array `floor` that's initialized to zeros. Read commands from an array that contains them. Keep track of the current turtle position at all times and whether the pen is currently up or down. Assume that the turtle always starts at position 0, 0 of the floor with its pen up. The set of turtle commands your program must process are shown in the following table:

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 50-by-50 array
`9`	End of data (sentinel)

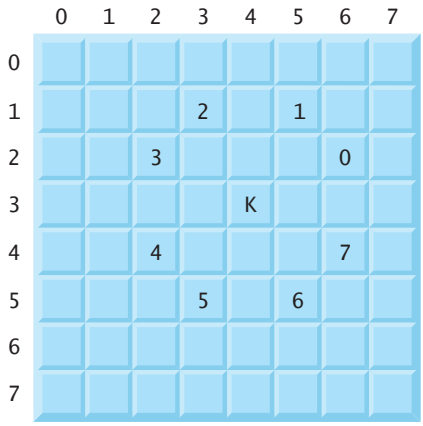
Suppose that the turtle is somewhere near the center of the floor. The following "program" would draw and print a 12-by-12 square:

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

As the turtle moves with the pen down, set elements of array `floor` to `1`s. When the `6` command is given, display an asterisk for each `1` in the array. For each 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 puzzles 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 (two in one direction and then 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 the following diagram:



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, and so on. 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? Were you close to the estimate?

b) Now let's develop a program that will move the knight around a chessboard. The board itself is represented by an 8-by-8 two-dimensional array `board`. Each square is initialized to zero. We describe each of the eight possible moves in terms of both its horizontal and vertical components. For example, a move of type 0, as shown in the preceding diagram, 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 one-dimensional arrays, `horizontal` and `vertical`, as follows:

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

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 have 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 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 develop an "accessibility heuristic" by classifying each square according to how accessible it is and always moving the knight to the square (within the knight's L-shaped moves, of course) that's most inaccessible. We label a two-dimensional 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? (*Optional:* Modify the program to run 64 tours, one 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.

6.25 ***(Knight's Tour: Brute-Force Approaches)*** In Exercise 6.24, we developed a Knight's Tour solution. 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 many problems with sheer computer power and relatively unsophisticated algorithms. Let's call this approach brute-force problem solving.

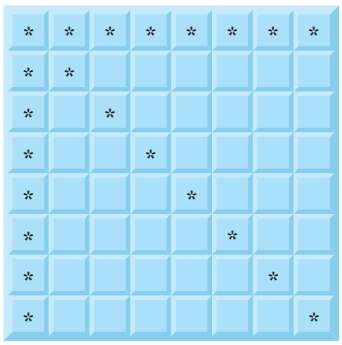
a) Use random numbers to enable the knight to walk at random around the chessboard in its legitimate L-shaped moves. Your program should run one tour and print the final chessboard. How far did the knight get?

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 a 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 could run for hours on a powerful computer.] Once again, track 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 Knight's Tour brute-force version with the accessibility-heuristic version. Which required more careful study of the problem? Which 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? Could we be certain (in advance) of obtaining a full tour with brute force? Argue the pros and cons of brute-force problem solving in general.

6.26 ***(Eight Queens)*** Another puzzle 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's 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 illustrated in the following diagram:



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 exercise, you'll 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 Exercise 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 Eight Queens 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.

6.29 ***(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 chessboard 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 6.24 to test for a closed tour if a full tour has occurred.

6.30 ***(The Sieve of Eratosthenes)*** A prime integer is any integer greater than 1 that can be divided evenly only by itself and 1. In this exercise, you'll use the Sieve of Eratosthenes to find all the prime numbers less than 1000. It works as follows:

a) Create a 100-element 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.
b) Starting with subscript 2 (1 is not prime), 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, and so on). 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, and so on).

When this process is complete, the array elements that are still set to 1 indicate that the subscript is a prime number. Write a program that determines and prints the prime numbers between 1 and 999. Ignore element 0 of the array.

Recursion Exercises

6.31 ***(Palindromes)*** A palindrome is a string that's spelled the same way forward and backward. Some examples of palindromes are: "radar," "able was i ere i saw elba," and, if you ignore blanks, "a man a plan a canal panama." Write a recursive function `testPalindrome` that returns 1 if the string stored in the array is a palindrome and 0 otherwise. The function should ignore spaces and punctuation in the string.

6.32 ***(Linear Search)*** Modify the program of Fig. 6.14 to use a recursive `linearSearch` function to perform the linear search of the array. The function should receive an integer array, the size of the array and the search key as arguments. If the search key is found, return the array subscript; otherwise, return –1.

6.33 ***(Binary Search)*** Modify the program of Fig. 6.15 to use a recursive `binarySearch` function to perform the binary search of the array. The function should receive an integer array, the starting subscript, the ending subscript and the search key as arguments. If the search key is found, return the array subscript; otherwise, return –1.

6.34 ***(Eight Queens)*** Modify the Eight Queens program you created in Exercise 6.26 to solve the problem recursively.

6.35 ***(Print an Array)*** Write a recursive function `printArray` that takes an array and the size of the array as arguments, prints the array, and returns nothing. The function should stop processing and return when it receives an array of size zero.

6.36 ***(Print a String Backward)*** Write a recursive function `stringReverse` that takes a character array as an argument, prints it back-to-front and returns nothing. The function should stop processing and return when the terminating null character of the string is encountered.

6.37 ***(Find the Minimum Value in an Array)*** Write a recursive function `recursiveMinimum` that takes an integer array and the array size as arguments and returns the smallest element of the array. The function should stop processing and return when it receives an array of one element.