Outline

6.1 Introduction
6.2 Arrays
6.3 Defining Arrays
6.4 Array Examples
6.5 Passing Arrays to Functions
6.6 Sorting Arrays
6.7 Case Study: Computing Mean, Median and Mode Using Arrays
6.8 Searching Arrays
6.9 Multidimensional Arrays
6.10 Variable-Length Arrays
6.11 Secure C Programming

Summary | *Terminology* | *Self-Review Exercises* | *Answers to Self-Review Exercises* | *Exercises* *Recursion Exercises*

6.1 Introduction

This chapter serves as an introduction to data structures. Arrays are data structures consisting of related data items of the same type. In Chapter 10, we discuss C's notion of `struct` (structure)—a data structure consisting of related data items of possibly different types. Arrays and structures are "static" entities in that they remain the same size throughout program execution (they may, of course, be of automatic storage class and hence created and destroyed each time the blocks in which they're defined are entered and exited).

6.2 Arrays

An array is a group of *contiguous* memory locations that all have the *same type*. To refer to a particular location or element in the array, we specify the array's name and the **position number** of the particular element in the array.

Figure 6.1 shows an integer array called `c`, containing 12 **elements**. Any one of these elements may be referred to by giving the array's name followed by the *position number* of the particular element in square brackets (`[]`). The first element in every array is the **zeroth element**. An array name, like other variable names, can contain only letters, digits and underscores and cannot begin with a digit.

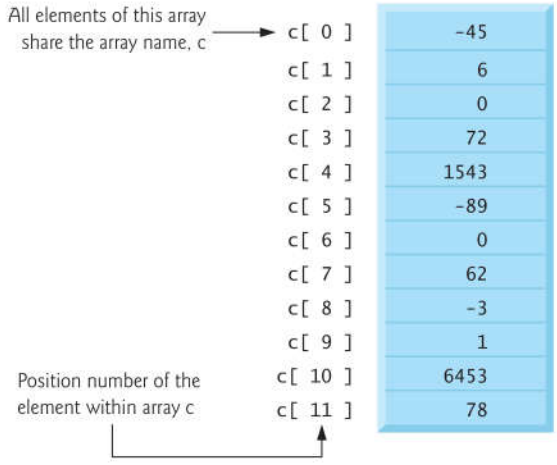

Fig. 6.1 | 12-element array.

The position number within square brackets is called a **subscript**. A subscript must be an integer or an integer expression. For example, if `a = 5` and `b = 6`, then the statement

```
c[ a + b ] += 2;
```

adds 2 to array element `c[11]`. A subscripted array name is an *lvalue*—it can be used on the left side of an assignment.

Let's examine array `c` (Fig. 6.1) more closely. The array's **name** is `c`. Its 12 elements are referred to as `c[0]`, `c[1]`, `c[2]`, …, `c[10]` and `c[11]`. The **value** stored in `c[0]` is `-45`, the value of `c[1]` is `6`, `c[2]` is `0`, `c[7]` is `62` and `c[11]` is `78`. To print the sum of the values contained in the first three elements of array `c`, we'd write

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

To divide the value of element 6 of array `c` by `2` and assign the result to the variable `x`, write

```
x = c[ 6 ] / 2;
```

The brackets used to enclose the subscript of an array are actually considered to be an *operator* in C. They have the same level of precedence as the *function call operator* (i.e., the parentheses that are placed after a function name to call that function). Figure 6.2 shows the precedence and associativity of the operators introduced to this point in the text.

Operators	Associativity	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

Fig. 6.2 | Operator precedence and associativity.

6.3 Defining Arrays

Arrays occupy space in memory. You specify the type of each element and the number of elements each array requires so that the computer may reserve the appropriate amount of memory. The following definition reserves 12 elements for integer array `c`, which has subscripts in the range 0–11.

```
int c[ 12 ];
```

The definition

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

reserves 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.

Arrays may contain other data types. For example, an array of type `char` 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.

6.4 Array Examples

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

Defining an Array and Using a Loop to Initialize the Array's Elements

Like any other variables, uninitialized array elements contain garbage values. Figure 6.3 uses `for` statements to initialize the elements of a 10-element integer array `n` to zeros and print the array in tabular format. The first `printf` statement (line 16) displays the column heads for the two columns printed in the subsequent `for` statement.

Notice that the variable `i` is declared to be of type `size_t` (line 9), which according to the C standard represents an unsigned integral type. This type is recommended for any variable that represents an array's size or an array's subscripts. Type `size_t` is defined in header `<stddef.h>`, which is often included by other headers (such as `<stdio.h>`). [*Note:* If you attempt to compile Fig. 6.3 and receive errors, simply include `<stddef.h>` in your program.]

```
1  // Fig. 6.3: fig06_03.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  {
8     int n[ 10 ]; // n is an array of 10 integers
9     size_t i; // counter
10
11    // initialize elements of array n to 0
12    for ( i = 0; i < 10; ++i ) {
13       n[ i ] = 0; // set element at location i to 0
14    } // end for
15
16    printf( "%s%13s\n", "Element", "Value" );
17
18    // output contents of array n in tabular format
19    for ( i = 0; i < 10; ++i ) {
20       printf( "%7u%13d\n", i, n[ i ] );
21    } // end for
22 } // end main
```

Fig. 6.3 | Initializing the elements of an array to zeros. (Part 1 of 2.)

```
Element         Value
      0             0
      1             0
      2             0
      3             0
      4             0
      5             0
      6             0
      7             0
      8             0
      9             0
```

Fig. 6.3 | Initializing the elements of an array to zeros. (Part 2 of 2.)

Initializing an Array in a Definition with an Initializer List

The elements of an array can also be initialized when the array is defined by following the definition with an equals sign and braces, {}, containing a comma-separated list of **array initializers**. Figure 6.4 initializes an integer array with 10 values (line 9) and prints the array in tabular format.

```
1  // Fig. 6.4: fig06_04.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  {
8     // use initializer list to initialize array n
9     int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
10    size_t i; // counter
11
12    printf( "%s%13s\n", "Element", "Value" );
13
14    // output contents of array in tabular format
15    for ( i = 0; i < 10; ++i ) {
16       printf( "%7u%13d\n", i, n[ i ] );
17    } // end for
18 } // end main
```

```
Element         Value
      0            32
      1            27
      2            64
      3            18
      4            95
      5            14
      6            90
      7            70
      8            60
      9            37
```

Fig. 6.4 | Initializing the elements of an array with an initializer list.

If there are *fewer* initializers than elements in the array, the remaining elements are initialized to zero. For example, the elements of the array `n` in Fig. 6.3 could have been initialized to zero as follows:

```
int n[ 10 ] = { 0 }; // initializes entire array to zeros
```

This *explicitly* initializes the first element to zero and initializes the remaining nine elements to zero because there are fewer initializers than there are elements in the array. It's important to remember that arrays are *not* automatically initialized to zero. You must at least initialize the first element to zero for the remaining elements to be automatically zeroed. Array elements are initialized before program startup for `static` arrays and at runtime for *automatic* arrays.

Common Programming Error 6.1

Forgetting to initialize the elements of an array.

The array definition

```
int n[ 5 ] = { 32, 27, 64, 18, 95, 14 };
```

causes a syntax error because there are six initializers and *only* five array elements.

Common Programming Error 6.2

Providing more initializers in an array initializer list than there are elements in the array is a syntax error.

If the array size is *omitted* from a definition with an initializer list, the number of elements in the array will be the number of elements in the initializer list. For example,

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

would create a five-element array initialized with the indicated values.

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

Figure 6.5 initializes the elements of a 10-element array `s` to the values `2`, `4`, `6`, …, `20` and prints the array in tabular format. The values are generated by multiplying the loop counter by `2` and adding `2`.

```
1  // Fig. 6.5: fig06_05.c
2  // Initializing the elements of array s to the even integers from 2 to 20.
3  #include <stdio.h>
4  #define SIZE 10 // maximum size of array
5
6  // function main begins program execution
7  int main( void )
8  {
9     // symbolic constant SIZE can be used to specify array size
10    int s[ SIZE ]; // array s has SIZE elements
11    size_t j; // counter
```

Fig. 6.5 | Initialize the elements of array `s` to the even integers from 2 to 20. (Part 1 of 2.)

```
12
13      for ( j = 0; j < SIZE; ++j ) { // set the values
14         s[ j ] = 2 + 2 * j;
15      } // end for
16
17      printf( "%s%13s\n", "Element", "Value" );
18
19      // output contents of array s in tabular format
20      for ( j = 0; j < SIZE; ++j ) {
21         printf( "%7u%13d\n", j, s[ j ] );
22      } // end for
23   } // end main
```

```
Element         Value
      0             2
      1             4
      2             6
      3             8
      4            10
      5            12
      6            14
      7            16
      8            18
      9            20
```

Fig. 6.5 | Initialize the elements of array `s` to the even integers from 2 to 20. (Part 2 of 2.)

The **`#define` preprocessor directive** is introduced in this program. Line 4

```
#define SIZE 10
```

defines a **symbolic constant** `SIZE` whose value is `10`. A symbolic constant is an identifier that's replaced with **replacement text** by the C preprocessor before the program is compiled. When the program is preprocessed, all occurrences of the symbolic constant `SIZE` are replaced with the replacement text `10`. Using symbolic constants to specify array sizes makes programs more **scalable**. In Fig. 6.5, we could have the first `for` loop (line 13) fill a 1000-element array by simply changing the value of `SIZE` in the `#define` directive from `10` to `1000`. If the symbolic constant `SIZE` had not been used, we'd have to change the program in *three* separate places. As programs get larger, this technique becomes more useful for writing clear, maintainable programs.

Common Programming Error 6.3

Ending a `#define` or `#include` preprocessor directive with a semicolon. Remember that preprocessor directives are not *C statements.*

If the `#define` preprocessor directive in line 4 is terminated with a semicolon, the preprocessor replaces all occurrences of the symbolic constant `SIZE` in the program with the text `10;`. This may lead to syntax errors at compile time, or logic errors at execution time. Remember that the preprocessor is *not* the C compiler.

Software Engineering Observation 6.1

Defining the size of each array as a symbolic constant makes programs more scalable.

Common Programming Error 6.4

Assigning a value to a symbolic constant in an executable statement is a syntax error. A symbolic constant is not a variable. The compiler does not reserve space for symbolic constants as it does for variables that hold values at execution time.

Good Programming Practice 6.1

Use only uppercase letters for symbolic constant names. This makes these constants stand out in a program and reminds you that symbolic constants are not *variables.*

Good Programming Practice 6.2

In multiword symbolic constant names, separate the words with underscores for readability.

Summing the Elements of an Array

Figure 6.6 sums the values contained in the 12-element integer array a. The `for` statement's body (line 16) does the totaling.

```
1  // Fig. 6.6: fig06_06.c
2  // Computing the sum of the elements of an array.
3  #include <stdio.h>
4  #define SIZE 12
5
6  // function main begins program execution
7  int main( void )
8  {
9     // use an initializer list to initialize the array
10    int a[ SIZE ] = { 1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45 };
11    size_t i; // counter
12    int total = 0; // sum of array
13
14    // sum contents of array a
15    for ( i = 0; i < SIZE; ++i ) {
16       total += a[ i ];
17    } // end for
18
19    printf( "Total of array element values is %d\n", total );
20 } // end main
```

```
Total of array element values is 383
```

Fig. 6.6 | Computing the sum of the elements of an array.

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.

> *Forty students were asked to rate the quality of the food in the student cafeteria on a scale of 1 to 10 (1 means awful and 10 means excellent). Place the 40 responses in an integer array and summarize the results of the poll.*

This is a typical array application (see Fig. 6.7). We wish to summarize the number of responses of each type (i.e., 1 through 10). The array `responses` (line 17) is a 40-element array of the students' responses. We use an 11-element array `frequency` (line 14) to count the number of occurrences of each response. 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.

Good Programming Practice 6.3

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.

Performance Tip 6.1

Sometimes performance considerations far outweigh clarity considerations.

```
1  // Fig. 6.7: fig06_07.c
2  // Analyzing a student poll.
3  #include <stdio.h>
4  #define RESPONSES_SIZE 40 // define array sizes
5  #define FREQUENCY_SIZE 11
6
7  // function main begins program execution
8  int main( void )
9  {
10    size_t answer; // counter to loop through 40 responses
11    size_t rating; // counter to loop through frequencies 1-10
12
13    // initialize frequency counters to 0
14    int frequency[ FREQUENCY_SIZE ] = { 0 };
15
16    // place the survey responses in the responses array
17    int responses[ RESPONSES_SIZE ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
18       1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
19       5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
20
21    // for each answer, select value of an element of array responses
22    // and use that value as subscript in array frequency to
23    // determine element to increment
24    for ( answer = 0; answer < RESPONSES_SIZE; ++answer ) {
25       ++frequency[ responses [ answer ] ];
26    } // end for
27
28    // display results
29    printf( "%s%17s\n", "Rating", "Frequency" );
30
31    // output the frequencies in a tabular format
32    for ( rating = 1; rating < FREQUENCY_SIZE; ++rating ) {
33       printf( "%6d%17d\n", rating, frequency[ rating ] );
34    } // end for
35 } // end main
```

Fig. 6.7 | Analyzing a student poll. (Part 1 of 2.)

```
Rating        Frequency
     1                2
     2                2
     3                2
     4                2
     5                5
     6               11
     7                5
     8                7
     9                1
    10                3
```

Fig. 6.7 | Analyzing a student poll. (Part 2 of 2.)

The `for` loop (line 24) takes the responses one at a time from the array `responses` and increments one of the 10 counters (`frequency[1]` to `frequency[10]`) in the `frequency` array. The key statement in the loop is line 25

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

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

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

which increments array element one. When `answer` is `1`, `responses[answer]` is `2`, so `++frequency[responses[answer]];` is interpreted as

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

which increments array element two. When `answer` is `2`, `responses[answer]` is `6`, so `++frequency[responses[answer]];` is interpreted as

```
++frequency[ 6 ];
```

which increments array element six, and so on. Regardless of the number of responses processed in the survey, only an 11-element array is required (ignoring element zero) to summarize the results. If the data contained invalid values such as 13, the program would attempt to add 1 to `frequency[13]`. This would be outside the bounds of the array. *C has no array bounds checking to prevent the program from referring to an element that does not exist.* Thus, an executing program can "walk off" either end of an array without warning—a security problem that we discuss in Section 6.11. You should ensure that all array references remain within the bounds of the array.

Common Programming Error 6.5

Referring to an element outside the array bounds.

Error-Prevention Tip 6.1

When looping through an array, the array subscript should never go below 0 and should always be less than the total number of elements in the array (size – 1). Make sure the loop-terminating condition prevents accessing elements outside this range.

Error-Prevention Tip 6.2

Programs should validate the correctness of all input values to prevent erroneous information from affecting a program's calculations.

Graphing Array Element Values with Histograms

Our next example (Fig. 6.8) reads numbers from an array and graphs the information in the form of a bar chart or histogram—each number is printed, then a bar consisting of that many asterisks is printed beside the number. The nested `for` statement (line 20) draws the bars. Note the use of `puts( "" )` to end each histogram bar (line 24).

```
1  // Fig. 6.8: fig06_08.c
2  // Displaying a histogram.
3  #include <stdio.h>
4  #define SIZE 10
5
6  // function main begins program execution
7  int main( void )
8  {
9     // use initializer list to initialize array n
10    int n[ SIZE ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
11    size_t i; // outer for counter for array elements
12    int j; // inner for counter counts *s in each histogram bar
13
14    printf( "%s%13s%17s\n", "Element", "Value", "Histogram" );
15
16    // for each element of array n, output a bar of the histogram
17    for ( i = 0; i < SIZE; ++i ) {
18       printf( "%7u%13d          ", i, n[ i ]) ;
19
20       for ( j = 1; j <= n[ i ]; ++j ) { // print one bar
21          printf( "%c", '*' );
22       } // end inner for
23
24       puts( "" ); // end a histogram bar
25    } // end outer for
26 } // end main
```

```
Element         Value         Histogram
      0            19         *******************
      1             3         ***
      2            15         ***************
      3             7         *******
      4            11         ***********
      5             9         *********
      6            13         *************
      7             5         *****
      8            17         *****************
      9             1         *
```

Fig. 6.8 | Displaying a histogram.

Rolling a Die 6,000,000 Times and Summarizing the Results in an Array

In Chapter 5, we stated that we'd show a more elegant method of writing the dice-rolling program of Fig. 5.12. The problem was to roll a single six-sided die 6,000,000 times to test whether the random number generator actually produces random numbers. An array version of this program is shown in Fig. 6.9.

```
1  // Fig. 6.9: fig06_09.c
2  // Roll a six-sided die 6,000,000 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 {
11    size_t face; // random die value 1 - 6
12    unsigned int roll; // roll counter 1-6,000,000
13    unsigned int frequency[ SIZE ] = { 0 }; // clear counts
14
15    srand( time( NULL ) ); // seed random number generator
16
17    // roll die 6,000,000 times
18    for ( roll = 1; roll <= 6000000; ++roll ) {
19       face = 1 + rand() % 6;
20       ++frequency[ face ]; // replaces entire switch of Fig. 5.8
21    } // end for
22
23    printf( "%s%17s\n", "Face", "Frequency" );
24
25    // output frequency elements 1-6 in tabular format
26    for ( face = 1; face < SIZE; ++face ) {
27       printf( "%4d%17d\n", face, frequency[ face ] );
28    } // end for
29 } // end main
```

```
Face        Frequency
   1           999753
   2          1000773
   3           999600
   4           999786
   5          1000552
   6           999536
```

Fig. 6.9 | Roll a six-sided die 6,000,000 times.

Using Character Arrays to Store and Manipulate Strings

We've discussed only integer arrays. However, arrays are capable of holding data of *any* 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 in C.

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 size of array `string1` is determined by the compiler based on the length of the string. The string `"first"` contains five characters *plus* a special *string-termination character* called the **null character**. Thus, array `string1` actually contains *six* elements. The character constant representing the null character is `'\0'`. All strings in C end with this character. 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, but this can be tedious. The preceding definition is equivalent to

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

Because a string is really an array of characters, we can access individual characters in a string directly using array subscript notation. For example, `string1[0]` is the character `'f'` and `string1[3]` is the character `'s'`.

We also can input a string directly into a character array from the keyboard using `scanf` and the conversion specifier `%s`. For example,

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

creates a character array capable of storing a string of *at most 19 characters* and a *terminating null character*. The statement

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

reads a string from the keyboard into `string2`. The name of the array is passed to `scanf` without the preceding `&` used with nonstring 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 6.5, when we discuss passing arrays to functions, we'll see that the value of an array name *is the address of the start of the array*; therefore, the `&` is not necessary. Function `scanf` 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. For this reason, we used the conversion specifier `%19s` so that `scanf` reads a maximum of 19 characters and does not write characters into memory beyond the end of the array `string2`.

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. Thus, `scanf` can write beyond the end of the array.

A character array representing a string can be output with `printf` and the `%s` conversion specifier. The array `string2` is printed with the statement

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

Function `printf`, like `scanf`, does *not* check how large the character array is. The characters of the string are printed until a terminating null character is encountered. [Consider what would print if, for some reason, the terminating null character were missing.]

Figure 6.10 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 individual characters of a string. The program uses a `for` statement (line 23) to loop through the `string1` array and print the individual characters separated by spaces, using the `%c` conversion specifier. The condition in the `for` statement is true while the counter is less than the size of the array and the terminating null character has *not* been encountered in the string. In this program, we read only strings that do not contain whitespace characters. We'll show how to read strings with whitespace characters in Chapter 8. Notice that lines 18–19 contain two string literals separated only by whitespace. The compiler automatically combines such string literals into one—this is helpful for making long string literals more readable.

```
1  // Fig. 6.10: fig06_10.c
2  // Treating character arrays as strings.
3  #include <stdio.h>
4  #define SIZE 20
5
6  // function main begins program execution
7  int main( void )
8  {
9     char string1[ SIZE ]; // reserves 20 characters
10    char string2[] = "string literal"; // reserves 15 characters
11    size_t i; // counter
12
13    // read string from user into array string1
14    printf( "%s", "Enter a string (no longer than 19 characters): " );
15    scanf( "%19s", string1 ); // input no more than 19 characters
16
17    // output strings
18    printf( "string1 is: %s\nstring2 is: %s\n"
19            "string1 with spaces between characters is:\n",
20            string1, string2 );
21
22    // output characters until null character is reached
23    for ( i = 0; i < SIZE && string1[ i ] != '\0'; ++i ) {
24       printf( "%c ", string1[ i ] );
25    } // end for
26
27    puts( "" );
28 } // end main
```

```
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. 6.10 | Treating character arrays as strings.

Static Local Arrays and Automatic Local Arrays

Chapter 5 discussed the storage-class specifier `static`. A `static` local variable exists for the *duration* of the program but is *visible* only in the function body. We can apply `static` to a local array definition so the array is *not* created and initialized each time the function

is called and the array is *not* destroyed each time the function is exited in the program. This reduces program execution time, particularly for programs with frequently called functions that contain large arrays.

Performance Tip 6.2

In functions that contain automatic arrays where the function is in and out of scope frequently, make the array `static` *so it's not created each time the function is called.*

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.

Figure 6.11 demonstrates function `staticArrayInit` (lines 21–40) with a local `static` array (line 24) and function `automaticArrayInit` (lines 43–62) with a local automatic array (line 46). Function `staticArrayInit` is called twice (lines 12 and 16). The local `static` array in the function is initialized to zero before program startup (line 24). 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 function call.

Function `automaticArrayInit` is also called twice (lines 13 and 17). The elements of the automatic local array in the function are initialized with the values 1, 2 and 3 (line 46). 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.

Common Programming Error 6.6

Assuming that elements of a local `static` *array are initialized to zero every time the function in which the array is defined is called.*

```
1  // Fig. 6.11: fig06_11.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 {
11    puts( "First call to each function:" );
12    staticArrayInit();
13    automaticArrayInit();
14
15    puts( "\n\nSecond call to each function:" );
16    staticArrayInit();
17    automaticArrayInit();
18 } // end main
19
20 // function to demonstrate a static local array
21 void staticArrayInit( void )
22 {
```

Fig. 6.11 | Static arrays are initialized to zero if not explicitly initialized. (Part 1 of 3.)

```
23     // initializes elements to 0 first time function is called
24     static int array1[ 3 ];
25     size_t i; // counter
26
27     puts( "\nValues on entering staticArrayInit:" );
28
29     // output contents of array1
30     for ( i = 0; i <= 2; ++i ) {
31        printf( "array1[ %u ] = %d  ", i, array1[ i ] );
32     } // end for
33
34     puts( "\nValues on exiting staticArrayInit:" );
35
36     // modify and output contents of array1
37     for ( i = 0; i <= 2; ++i ) {
38        printf( "array1[ %u ] = %d  ", i, array1[ i ] += 5 );
39     } // end for
40  } // end function staticArrayInit
41
42  // function to demonstrate an automatic local array
43  void automaticArrayInit( void )
44  {
45     // initializes elements each time function is called
46     int array2[ 3 ] = { 1, 2, 3 };
47     size_t i; // counter
48
49     puts( "\n\nValues on entering automaticArrayInit:" );
50
51     // output contents of array2
52     for ( i = 0; i <= 2; ++i ) {
53        printf("array2[ %u ] = %d  ", i, array2[ i ] );
54     } // end for
55
56     puts( "\nValues on exiting automaticArrayInit:" );
57
58     // modify and output contents of array2
59     for ( i = 0; i <= 2; ++i ) {
60        printf( "array2[ %u ] = %d  ", i, array2[ i ] += 5 );
61     } // end for
62  } // end function automaticArrayInit
```

```
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
```

Fig. 6.11 | Static arrays are initialized to zero if not explicitly initialized. (Part 2 of 3.)

```
Second call to each function:

Values on entering staticArrayInit:
array1[ 0 ] = 5  array1[ 1 ] = 5  array1[ 2 ] = 5
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 on exiting automaticArrayInit:
array2[ 0 ] = 6  array2[ 1 ] = 7  array2[ 2 ] = 8
```

Fig. 6.11 | Static arrays are initialized to zero if not explicitly initialized. (Part 3 of 3.)

6.5 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*. C automatically passes arrays to functions *by reference* (again, we'll see in Chapter 7 that this is *not* a contradiction)—the called functions can modify the element values in the callers' original arrays. The name of the array evaluates to the address of the first element of the array. Because the starting address of the array is passed, the called function knows precisely where the array is stored. Therefore, when the called function modifies array elements in its function body, it's modifying the actual elements of the array in their *original* memory locations.

Figure 6.12 demonstrates that an array name is really the *address* of the first element of the array by printing `array`, `&array[0]` and `&array` using the **`%p` conversion specifier**—a special conversion specifier for printing addresses. The `%p` conversion specifier 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 (these letters are the hexadecimal equivalents of the decimal numbers 10–15). Appendix C 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, namely `0012FF78`. The output of this program is system dependent, but the addresses are always identical for a particular execution of this program on a particular computer.

Performance Tip 6.3

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.*

```
1  // Fig. 6.12: fig06_12.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  {
8     char array[ 5 ]; // define an array of size 5
9
10    printf( "    array = %p\n&array[0] = %p\n   &array = %p\n",
11       array, &array[ 0 ], &array );
12 } // end main
```

```
    array = 0012FF78
&array[0] = 0012FF78
   &array = 0012FF78
```

Fig. 6.12 | Array name is the same as the address of the array's first element.

Software Engineering Observation 6.2

It's possible to pass an array by value (by using a simple trick we explain in Chapter 10).

Although entire arrays are passed by reference, individual array elements are passed by *value* exactly as simple variables are. Such simple single pieces of data (such as individual `int`s, `float`s and `char`s) are called scalars. To pass an element of an array to a function, use the subscripted name of the array element 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.

For a function to receive an array through a function call, the function's parameter list *must* specify that an array will be received. For example, the function header for function `modifyArray` (that we called earlier in this section) might be written as

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

indicating that `modifyArray` expects to receive an array of integers in parameter `b` and the number of array elements in parameter `size`. The *size* of the array is *not* required between the array brackets. If it's included, the compiler checks that it's greater than zero, then ignores it. Specifying a negative size is a compilation error. Because arrays are automatically passed by reference, when the called function uses the array name `b`, it will be referring to the array in the caller (array `hourlyTemperatures` in the preceding call). In Chapter 7, we introduce other notations for indicating that an array is being received by a function. As we'll see, these notations are based on the intimate relationship between arrays and pointers in C.

Difference Between Passing an Entire Array and Passing an Array Element

Figure 6.13 demonstrates the difference between passing an entire array and passing an array element. The program first prints the five elements of integer array `a` (lines 20–22). Next, `a` and its size are passed to function `modifyArray` (line 27), where each of `a`'s ele-

ments is multiplied by 2 (lines 53–55). Then a is reprinted in `main` (lines 32–34). As the output shows, the elements of a are indeed modified by `modifyArray`. Now the program prints the value of `a[3]` (line 38) and passes it to function `modifyElement` (line 40). Function `modifyElement` multiplies its argument by 2 (line 63) and prints the new value. When `a[3]` is reprinted in `main` (line 43), it has not been modified, because individual array elements are passed by value.

There may be situations in your programs in which a function should *not* be allowed to modify array elements. C provides the type qualifier **`const`** (for "constant") that can be used to prevent modification of array values in a function. When an array parameter is preceded by the `const` qualifier, the array elements become constant in the function body, and any attempt to modify an element of the array in the function body results in a compile-time error. This enables you to correct a program so it does not attempt to modify array elements.

```
1  // Fig. 6.13: fig06_13.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 {
13    int a[ SIZE ] = { 0, 1, 2, 3, 4 }; // initialize array a
14    size_t i; // counter
15
16    puts( "Effects of passing entire array by reference:\n\nThe "
17       "values of the original array are:" );
18
19    // output original array
20    for ( i = 0; i < SIZE; ++i ) {
21       printf( "%3d", a[ i ] );
22    } // end for
23
24    puts( "" );
25
26    // pass array a to modifyArray by reference
27    modifyArray( a, SIZE );
28
29    puts( "The values of the modified array are:" );
30
31    // output modified array
32    for ( i = 0; i < SIZE; ++i ) {
33       printf( "%3d", a[ i ] );
34    } // end for
35
```

Fig. 6.13 | Passing arrays and individual array elements to functions. (Part 1 of 2.)

```
36    // output value of a[ 3 ]
37    printf( "\n\n\nEffects of passing array element "
38       "by value:\n\nThe value of a[3] is %d\n", a[ 3 ] );
39
40    modifyElement( a[ 3 ] ); // pass array element a[ 3 ] by value
41
42    // output value of a[ 3 ]
43    printf( "The value of a[ 3 ] is %d\n", a[ 3 ] );
44 } // end main
45
46 // in function modifyArray, "b" points to the original array "a"
47 // in memory
48 void modifyArray( int b[], size_t size )
49 {
50    size_t j; // counter
51
52    // multiply each array element by 2
53    for ( j = 0; j < size; ++j ) {
54       b[ j ] *= 2; // actually modifies original array
55    } // end for
56 } // end function modifyArray
57
58 // in function modifyElement, "e" is a local copy of array element
59 // a[ 3 ] passed from main
60 void modifyElement( int e )
61 {
62    // multiply parameter by 2
63    printf( "Value in modifyElement is %d\n", e *= 2 );
64 } // end function modifyElement
```

```
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. 6.13 | Passing arrays and individual array elements to functions. (Part 2 of 2.)

Using the `const` *Qualifier with Array Parameters*

Figure 6.14 demonstrates the `const` qualifier. Function `tryToModifyArray` (line 19) is defined with parameter `const int b[]`, which specifies that array `b` is *constant* and *cannot* be modified. The output shows the error messages produced by the compiler—the errors may be different for your compiler. Each of the function's three attempts to modify array elements results in the compiler error "`l-value specifies a const object`." The `const` qualifier is discussed in additional contexts in Chapter 7.

```c
// Fig. 6.14: fig06_14.c
// Using the const type qualifier with arrays.
#include <stdio.h>

void tryToModifyArray( const int b[] ); // function prototype

// function main begins program execution
int main( void )
{
   int a[] = { 10, 20, 30 }; // initialize array a

   tryToModifyArray( a );

   printf("%d %d %d\n", a[ 0 ], a[ 1 ], a[ 2 ] );
} // end main

// in function tryToModifyArray, array b is const, so it cannot be
// used to modify the original array a in main.
void tryToModifyArray( const int b[] )
{
   b[ 0 ] /= 2; // error
   b[ 1 ] /= 2; // error
   b[ 2 ] /= 2; // error
} // end function tryToModifyArray
```

```
fig06_14.c(21) : error C2166: l-value specifies const object
fig06_14.c(22) : error C2166: l-value specifies const object
fig06_14.c(23) : error C2166: l-value specifies const object
```

Fig. 6.14 | Using the `const` type qualifier with arrays.

Software Engineering Observation 6.3

The `const` type qualifier can be applied to an array parameter in a function definition to prevent the original array from being modified in the function body. 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.

6.6 Sorting Arrays

Sorting data (i.e., placing the data into a particular order such as ascending or descending) is one of the most important computing applications. A bank sorts all checks by account number so that it can 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 which has attracted some of the most intense research efforts in the field of computer science. In this chapter we discuss what is perhaps the simplest known sorting scheme. In Chapter 12 and Appendix E, we investigate more complex schemes that yield better performance.

Performance Tip 6.4

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.

Figure 6.15 sorts the values in the elements of the 10-element array a (line 10) into ascending order. The technique we use is called the **bubble sort** 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 (element 0 and element 1, then element 1 and element 2, etc.) are compared. 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, their values are swapped in the array.

```
1  // Fig. 6.15: fig06_15.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  {
9     // initialize a
10    int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
11    int pass; // passes counter
12    size_t i; // comparisons counter
13    int hold; // temporary location used to swap array elements
14
15    puts( "Data items in original order" );
16
17    // output original array
18    for ( i = 0; i < SIZE; ++i ) {
19       printf( "%4d", a[ i ] );
20    } // end for
21
22    // bubble sort
23    // loop to control number of passes
24    for ( pass = 1; pass < SIZE; ++pass ) {
25
26       // loop to control number of comparisons per pass
27       for ( i = 0; i < SIZE - 1; ++i ) {
28
29          // compare adjacent elements and swap them if first
30          // element is greater than second element
31          if ( a[ i ] > a[ i + 1 ] ) {
32             hold = a[ i ];
33             a[ i ] = a[ i + 1 ];
34             a[ i + 1 ] = hold;
35          } // end if
36       } // end inner for
37    } // end outer for
```

Fig. 6.15 | Sorting an array's values into ascending order. (Part 1 of 2.)

```
38
39      puts( "\nData items in ascending order" );
40
41      // output sorted array
42      for ( i = 0; i < SIZE; ++i ) {
43         printf( "%4d", a[ i ] );
44      } // end for
45
46      puts( "" );
47   } // end main
```

```
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. 6.15 | 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. Because of the way the successive comparisons are made, a large value may move down the array many positions on a single pass, but a small value may move up only one position.

On the first pass, the largest value is guaranteed to sink to the bottom element of the array, `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 of the array are needed to sort the array, even though there are *ten* elements.

The sorting is performed by the nested `for` loops (lines 24–37). If a swap is necessary, it's performed by the three assignments

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

where the extra 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`.

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 a few of these in the exercises. More advanced courses investigate sorting and searching in greater depth.

6.7 Case Study: Computing Mean, Median and Mode Using Arrays

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 6.16 uses 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. Figure 6.17 contains a sample run of this program. This example includes most of the common manipulations usually required in array problems, including passing arrays to functions.

```
1  // Fig. 6.16: fig06_16.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 unsigned int answer[] );
9  void median( unsigned int answer[] );
10 void mode( unsigned int freq[], unsigned const int answer[] ) ;
11 void bubbleSort( int a[] );
12 void printArray( unsigned const int a[] );
13
14 // function main begins program execution
15 int main( void )
16 {
17    unsigned int frequency[ 10 ] = { 0 }; // initialize array frequency
18
19    // initialize array response
20    unsigned int response[ SIZE ] =
21       { 6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
22         7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
23         6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
24         7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
25         6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
26         7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
27         5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
28         7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
29         7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
30         4, 5, 6, 1, 6, 5, 7, 8, 7 };
31
32    // process responses
33    mean( response );
34    median( response );
35    mode( frequency, response );
36 } // end main
37
38 // calculate average of all response values
39 void mean( const unsigned int answer[] )
40 {
```

Fig. 6.16 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 1 of 4.)

```
41     size_t j; // counter for totaling array elements
42     unsigned int total = 0; // variable to hold sum of array elements
43
44     printf( "%s\n%s\n%s\n", "********", "  Mean", "********" );
45
46     // total response values
47     for ( j = 0; j < SIZE; ++j ) {
48        total += answer[ j ];
49     } // end for
50
51     printf( "The mean is the average value of the data\n"
52             "items. The mean is equal to the total of\n"
53             "all the data items divided by the number\n"
54             "of data items ( %u ). The mean value for\n"
55             "this run is: %u / %u = %.4f\n\n",
56             SIZE, total, SIZE, (  double  ) total / SIZE );
57  } // end function mean
58
59  // sort array and determine median element's value
60  void median( unsigned int answer[] )
61  {
62     printf( "\n%s\n%s\n%s\n%s",
63             "********", " Median", "********",
64             "The unsorted array of responses is" );
65
66     printArray( answer ); // output unsorted array
67
68     bubbleSort( answer ); // sort array
69
70     printf( "%s", "\n\nThe sorted array is" );
71     printArray( answer ); // output sorted array
72
73     // display median element
74     printf( "\n\nThe median is element %u of\n"
75             "the sorted %u element array.\n"
76             "For this run the median is %u\n\n",
77             SIZE / 2, SIZE, answer[ SIZE / 2 ] );
78  } // end function median
79
80  // determine most frequent response
81  void mode( unsigned int freq[], const unsigned int answer[] )
82  {
83     size_t rating; // counter for accessing elements 1-9 of array freq
84     size_t j; // counter for summarizing elements 0-98 of array answer
85     unsigned int h; // counter for diplaying histograms freq array values
86     unsigned int largest = 0; // represents largest frequency
87     unsigned int modeValue = 0; // represents most frequent response
88
89     printf( "\n%s\n%s\n%s\n",
90             "********", "  Mode", "********" );
```

Fig. 6.16 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 2 of 4.)

```
91
92      // initialize frequencies to 0
93      for ( rating = 1; rating <= 9; ++rating ) {
94         freq[ rating ] = 0;
95      } // end for
96
97      // summarize frequencies
98      for ( j = 0; j < SIZE; ++j ) {
99         ++freq[ answer[ j ] ];
100     } // end for
101
102     // output headers for result columns
103     printf( "%s%11s%19s\n\n%54s\n%54s\n\n",
104             "Response", "Frequency", "Histogram",
105             "1    1    2    2", "5    0    5    0    5" );
106
107     // output results
108     for ( rating = 1; rating <= 9; ++rating ) {
109        printf( "%8u%11u          ", rating, freq[ rating ] );
110
111        // keep track of mode value and largest frequency value
112        if ( freq[ rating ] > largest ) {
113           largest = freq[ rating ];
114           modeValue = rating;
115        } // end if
116
117        // output histogram bar representing frequency value
118        for ( h = 1; h <= freq[ rating ]; ++h ) {
119           printf( "%s", "*" );
120        } // end inner for
121
122        puts( "" ); // being new line of output
123     } // end outer for
124
125     // display the mode value
126     printf( "\nThe mode is the most frequent value.\n"
127             "For this run the mode is %u which occurred"
128             " %u times.\n", modeValue, largest );
129  } // end function mode
130
131  // function that sorts an array with bubble sort algorithm
132  void bubbleSort( unsigned int a[] )
133  {
134     unsigned int pass; // pass counter
135     size_t j; // comparison counter
136     unsigned int hold; // temporary location used to swap elements
137
138     // loop to control number of passes
139     for ( pass = 1; pass < SIZE; ++pass ) {
140
```

Fig. 6.16 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 3 of 4.)

```
141        // loop to control number of comparisons per pass
142        for ( j = 0; j < SIZE - 1; ++j ) {
143
144           // swap elements if out of order
145           if ( a[ j ] > a[ j + 1 ] ) {
146              hold = a[ j ];
147              a[ j ] = a[ j + 1 ];
148              a[ j + 1 ] = hold;
149           } // end if
150        } // end inner for
151     } // end outer for
152  } // end function bubbleSort
153
154  // output array contents (20 values per row)
155  void printArray( const unsigned int a[] )
156  {
157     size_t j; // counter
158
159     // output array contents
160     for ( j = 0; j < SIZE; ++j ) {
161
162        if ( j % 20 == 0 ) { // begin new line every 20 values
163           puts( "" );
164        } // end if
165
166        printf( "%2u", a[ j ] );
167     } // end for
168  } // end function printArray
```

Fig. 6.16 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 4 of 4.)

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

Fig. 6.17 | Sample run for the survey data analysis program. (Part 1 of 2.)

```
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 6 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          Histogram

                                    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. 6.17 | Sample run for the survey data analysis program. (Part 2 of 2.)

Mean

The *mean* is the *arithmetic average* of the 99 values. Function `mean` (Fig. 6.16, lines 39–57) computes the mean by totaling the 99 elements and dividing the result by 99.

Median

The median is the *middle* value. Function `median` (lines 60–78) determines the median by calling function `bubbleSort` (defined in lines 132–152) to sort the array of responses into ascending order, then picking `answer[SIZE / 2]` (the middle element) of the sorted array. 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. Function `printArray` (lines 155–168) is called to output the `response` array.

Mode

The *mode* is the *value that occurs most frequently* among the 99 responses. Function `mode` (lines 81–129) 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 histogram to aid in determining the mode graphically.

6.8 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 particular element of an array is called searching. In this section we discuss two searching techniques—the simple linear search technique and the more efficient (but more complex) binary search technique. Exercise 6.32 and Exercise 6.33 ask you to implement *recursive* versions of the linear search and the binary search, respectively.

Searching an Array with Linear Search

The linear search (Fig. 6.18) compares each element of the array with the search key. 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. On average, therefore, the program will have to compare the search key with *half* the elements of the array.

```
1  // Fig. 6.18: fig06_18.c
2  // Linear search of an array.
3  #include <stdio.h>
4  #define SIZE 100
5
6  // function prototype
7  size_t linearSearch( const int array[], int key, size_t size );
8
9  // function main begins program execution
10 int main( void )
11 {
12    int a[ SIZE ]; // create array a
13    size_t x; // counter for initializing elements 0-99 of array a
14    int searchKey; // value to locate in array a
15    size_t element; // variable to hold location of searchKey or -1
16
17    // create some data
18    for ( x = 0; x < SIZE; ++x ) {
19       a[ x ] = 2 * x;
20    } // end for
21
22    puts( "Enter integer search key:" );
23    scanf( "%d", &searchKey );
24
25    // attempt to locate searchKey in array a
26    element = linearSearch( a, searchKey, SIZE );
27
28    // display results
29    if ( element != -1 ) {
30       printf( "Found value in element %d\n", element );
31    } // end if
32    else {
33       puts( "Value not found" );
34    } // end else
35 } // end main
```

Fig. 6.18 | Linear search of an array. (Part 1 of 2.)

```
36
37 // compare key to every element of array until the location is found
38 // or until the end of array is reached; return subscript of element
39 // if key is found or -1 if key is not found
40 size_t linearSearch( const int array[], int key, size_t size )
41 {
42    size_t n; // counter
43
44    // loop through array
45    for ( n = 0; n < size; ++n ) {
46
47       if ( array[ n ] == key ) {
48          return n; // return location of key
49       } // end if
50    } // end for
51
52    return -1; // key not found
53 } // end function linearSearch
```

```
Enter integer search key:
36
Found value in element 18
```

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

Fig. 6.18 | Linear search of an array. (Part 2 of 2.)

Searching an Array with Binary Search

The linear searching method 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 the elements in a sorted array after each comparison. 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 subscript of that element 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 element of the array, the *first half* of the array is searched, otherwise the *second half* is searched. If the search key is not found in the specified subarray (piece of the original array), the algorithm is repeated 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 (i.e., the search key is not found).

In a worst case-scenario, searching an 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. An 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 the linear search that required 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.

Figure 6.19 presents the *iterative* version of function `binarySearch` (lines 42–72). The function receives four arguments—an integer array `b` to be searched, an integer `searchKey`, the `low` array subscript and the `high` array subscript (these define the portion of the array to be searched). 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 `high` subscript is set to `middle - 1` and the search is continued on the elements from `low` to `middle - 1`. If the search key is *greater than* the middle element, the `low` subscript is set to `middle + 1` and the search is continued on the elements from `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. The program uses function `printHeader` (lines 75–94) to output the array subscripts and function `printRow` (lines 98–118) to output each subarray during the binary search process. The middle element in each subarray is marked with an asterisk (*) to indicate the element to which the search key is compared.

```
1  // Fig. 6.19: fig06_19.c
2  // Binary search of a sorted array.
3  #include <stdio.h>
4  #define SIZE 15
5
6  // function prototypes
7  size_t binarySearch(const int b[], int searchKey, 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 {
14    int a[ SIZE ]; // create array a
15    size_t i; // counter for initializing elements of array a
16    int key; // value to locate in array a
17    size_t result; // variable to hold location of key or -1
18
19    // create data
20    for ( i = 0; i < SIZE; ++i ) {
21       a[ i ] = 2 * i;
22    } // end for
23
24    printf( "%s", "Enter a number between 0 and 28: " );
25    scanf( "%d", &key );
```

Fig. 6.19 | Binary search of a sorted array. (Part 1 of 4.)

```
26
27    printHeader();
28
29    // search for key in array a
30    result = binarySearch( a, key, 0, SIZE - 1 );
31
32    // display results
33    if ( result != -1 ) {
34       printf( "\n%d found in array element %d\n", key, result );
35    } // end if
36    else {
37       printf( "\n%d not found\n", key );
38    } // end else
39 } // end main
40
41 // function to perform binary search of an array
42 size_t binarySearch(const int b[], int searchKey, size_t low, size_t high)
43 {
44    int middle; // variable to hold middle element of array
45
46    // loop until low subscript is greater than high subscript
47    while ( low <= high ) {
48
49       // determine middle element of subarray being searched
50       middle = ( low + high ) / 2;
51
52       // display subarray used in this loop iteration
53       printRow( b, low, middle, high );
54
55       // if searchKey matched middle element, return middle
56       if ( searchKey == b[ middle ] ) {
57          return middle;
58       } // end if
59
60       // if searchKey less than middle element, set new high
61       else if ( searchKey < b[ middle ] ) {
62          high = middle - 1; // search low end of array
63       } // end else if
64
65       // if searchKey greater than middle element, set new low
66       else {
67          low = middle + 1; // search high end of array
68       } // end else
69    } // end while
70
71    return -1; // searchKey not found
72 } // end function binarySearch
73
74 // Print a header for the output
75 void printHeader( void )
76 {
77    unsigned int i; // counter
78
```

Fig. 6.19 | Binary search of a sorted array. (Part 2 of 4.)

```
79      puts( "\nSubscripts:" );
80
81      // output column head
82      for ( i = 0; i < SIZE; ++i ) {
83         printf( "%3u ", i );
84      } // end for
85
86      puts( "" ); // start new line of output
87
88      // output line of - characters
89      for ( i = 1; i <= 4 * SIZE; ++i ) {
90         printf( "%s", "-" );
91      } // end for
92
93      puts( "" ); // start new line of output
94   } // end function printHeader
95
96   // Print one row of output showing the current
97   // part of the array being processed.
98   void printRow( const int b[], size_t low, size_t mid, size_t high )
99   {
100     size_t i; // counter for iterating through array b
101
102     // loop through entire array
103     for ( i = 0; i < SIZE; ++i ) {
104
105        // display spaces if outside current subarray range
106        if ( i < low || i > high ) {
107           printf( "%s", "    " );
108        } // end if
109        else if ( i == mid ) { // display middle element
110           printf( "%3d*", b[ i ] ); // mark middle value
111        } // end else if
112        else { // display other elements in subarray
113           printf( "%3d ", b[ i ] );
114        } // end else
115     } // end for
116
117     puts( "" ); // start new line of output
118  } // end function printRow
```

```
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
```

Fig. 6.19 | Binary search of a sorted array. (Part 3 of 4.)

```
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 in array element 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 in array element 3
```

Fig. 6.19 | Binary search of a sorted array. (Part 4 of 4.)

6.9 Multidimensional Arrays

Arrays in C can have multiple subscripts. A common use of multiple-subscripted arrays, which the C standard refers to as **multidimensional arrays**, is to represent **tables** of values consisting of information arranged in *rows* and *columns*. To identify a particular table element, we must specify two subscripts: The *first* (by convention) identifies the element's *row* and the *second* (by convention) identifies the element's *column*. Tables or arrays that require two subscripts to identify a particular element are called **double-subscripted arrays**. Multidimensional arrays can have more than two subscripts.

Figure 6.20 illustrates a double-subscripted array, 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**.

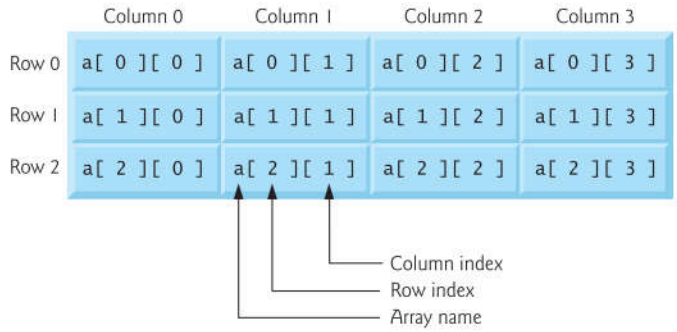

Fig. 6.20 | Double-subscripted array with three rows and four columns.

Every element in array a is identified in Fig. 6.20 by an element name of the form a[i][j]; a is the name of the array, and i and j are the subscripts that uniquely identify each element in a. The names of the elements in row 0 all have a first subscript of 0; the names of the elements in column 3 all have a second subscript of 3.

Common Programming Error 6.7

Referencing a double-subscripted array element as a[x, y] *instead of* a[x][y] *is a* logic *error. C interprets* a[x, y] *as* a[y] *(because the comma in this context is treated as a comma operator), so this programmer error is* not *a syntax error.*

A multidimensional array can be initialized when it's defined, much like a single-subscripted array. For example, a double-subscripted array int b[2][2] could be defined and initialized with

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

The values 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, the remaining elements of that row are initialized to 0.* Thus,

```
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 6.21 demonstrates defining and initializing double-subscripted arrays.

```
1  // Fig. 6.21: fig06_21.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  {
10    // initialize array1, array2, array3
11    int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
12    int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
13    int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };
14
15    puts( "Values in array1 by row are:" );
16    printArray( array1 );
17
18    puts( "Values in array2 by row are:" );
19    printArray( array2 );
20
21    puts( "Values in array3 by row are:" );
22    printArray( array3 );
23 } // end main
```

Fig. 6.21 | Initializing multidimensional arrays. (Part 1 of 2.)

```
24
25 // function to output array with two rows and three columns
26 void printArray( int a[][ 3 ] )
27 {
28    size_t i; // row counter
29    size_t j; // column counter
30
31    // loop through rows
32    for ( i = 0; i <= 1; ++i ) {
33
34       // output column values
35       for ( j = 0; j <= 2; ++j ) {
36          printf( "%d ", a[ i ][ j ] );
37       } // end inner for
38
39       printf( "\n" ); // start new line of output
40    } // end outer for
41 } // end function printArray
```

```
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. 6.21 | Initializing multidimensional arrays. (Part 2 of 2.)

The program defines three arrays of two rows and three columns (six elements each). The definition of `array1` (line 11) provides six initializers in two sublists. The first sublist initializes *row 0* of the array to the values 1, 2 and 3; and the second sublist initializes *row 1* of the array to the values 4, 5 and 6.

If the braces around each sublist are removed from the `array1` initializer list, the compiler initializes the elements of the first row followed by the elements of the second row. The definition of `array2` (line 12) provides five initializers. The initializers are assigned to the first row, then the second row. Any elements that do *not* have an explicit initializer are initialized to zero automatically, so `array2[1][2]` is initialized to 0.

The definition of `array3` (line 13) provides three initializers in two sublists. The sublist for the first row *explicitly* initializes the first two elements of the first row to 1 and 2. The third element is initialized to *zero*. The sublist for the second row explicitly initializes the first element to 4. The last two elements are initialized to *zero*.

The program calls `printArray` (lines 26–41) to output each array's elements. The function definition specifies the array parameter as `const int a[][3]`. When we receive a single-subscripted array as a parameter, the array brackets are *empty* in the function's parameter list. The first subscript of a multidimensional array is not required either, but all subsequent subscripts are required. The compiler uses these subscripts 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 double-subscripted 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 element in the array. In a double-subscripted array, each row is basically a single-subscripted 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. Thus, 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.

Many common array manipulations use `for` repetition statements. For example, the following statement sets all the elements in row 2 of array `a` in Fig. 6.20 to zero:

```
for ( 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 second (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;
```

The following nested `for` statement determines the total of all the elements in array `a`.

```
total = 0;

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

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

Two-Dimensonal Array Manipulations

Figure 6.22 performs several other common array manipulations on 3-by-4 array `studentGrades` using `for` statements. Each row of the array 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 41–60) determines the lowest grade of any student for the semester. Function `maximum` (lines 63–82) determines the highest grade of any student for the semester. Function `average` (lines 85–96) determines a particular student's semester average. Function `printArray` (lines 99–118) outputs the double-subscripted array in a neat, tabular format.

```
1  // Fig. 6.22: fig06_22.c
2  // Double-subscripted array manipulations.
3  #include <stdio.h>
4  #define STUDENTS 3
5  #define EXAMS 4
6
7  // function prototypes
8  int minimum( int grades[][ EXAMS ], size_t pupils, size_t tests );
9  int maximum( int grades[][ EXAMS ], size_t pupils, size_t tests );
10 double average( const int setOfGrades[], size_t tests );
11 void printArray( int grades[][ EXAMS ], size_t pupils, size_t tests );
12
13 // function main begins program execution
14 int main( void )
15 {
16    size_t student; // student counter
17
18    // initialize student grades for three students (rows)
19    int studentGrades[ STUDENTS ][ EXAMS ] =
20       { { 77, 68, 86, 73 },
21         { 96, 87, 89, 78 },
22         { 70, 90, 86, 81 } };
23
24    // output array studentGrades
25    puts( "The array is:" );
26    printArray( studentGrades, STUDENTS, EXAMS );
27
28    // determine smallest and largest grade values
29    printf( "\n\nLowest grade: %d\nHighest grade: %d\n",
30       minimum( studentGrades, STUDENTS, EXAMS ),
31       maximum( studentGrades, STUDENTS, EXAMS ) );
32
33    // calculate average grade for each student
34    for ( student = 0; student < STUDENTS; ++student ) {
35       printf( "The average grade for student %u is %.2f\n",
36          student, average( studentGrades[ student ], EXAMS ) );
37    } // end for
38 } // end main
39
40 // Find the minimum grade
41 int minimum( int grades[][ EXAMS ], size_t pupils, size_t tests )
42 {
43    size_t i; // student counter
44    size_t j; // exam counter
45    int lowGrade = 100; // initialize to highest possible grade
46
47    // loop through rows of grades
48    for ( i = 0; i < pupils; ++i ) {
49
50       // loop through columns of grades
51       for ( j = 0; j < tests; ++j ) {
52
53          if ( grades[ i ][ j ] < lowGrade ) {
```

Fig. 6.22 | Double-subscripted array manipulations. (Part 1 of 3.)

```
54              lowGrade = grades[ i ][ j ];
55           } // end if
56        } // end inner for
57     } // end outer for
58
59     return lowGrade; // return minimum grade
60  } // end function minimum
61
62  // Find the maximum grade
63  int maximum( int grades[][ EXAMS ], size_t pupils, size_t tests )
64  {
65     size_t i; // student counter
66     size_t j; // exam counter
67     int highGrade = 0; // initialize to lowest possible grade
68
69     // loop through rows of grades
70     for ( i = 0; i < pupils; ++i ) {
71
72        // loop through columns of grades
73        for ( j = 0; j < tests; ++j ) {
74
75           if ( grades[ i ][ j ] > highGrade ) {
76              highGrade = grades[ i ][ j ];
77           } // end if
78        } // end inner for
79     } // end outer for
80
81     return highGrade; // return maximum grade
82  } // end function maximum
83
84  // Determine the average grade for a particular student
85  double average( const int setOfGrades[], size_t tests )
86  {
87     size_t i; // exam counter
88     int total = 0; // sum of test grades
89
90     // total all grades for one student
91     for ( i = 0; i < tests; ++i ) {
92        total += setOfGrades[ i ];
93     } // end for
94
95     return ( double ) total / tests; // average
96  } // end function average
97
98  // Print the array
99  void printArray( int grades[][ EXAMS ], size_t pupils, size_t tests )
100 {
101    size_t i; // student counter
102    size_t j; // exam counter
103
104    // output column heads
105    printf( "%s", "                [0]  [1]  [2]  [3]" );
106
```

Fig. 6.22 | Double-subscripted array manipulations. (Part 2 of 3.)

```
107    // output grades in tabular format
108    for ( i = 0; i < pupils; ++i ) {
109
110       // output label for row
111       printf( "\nstudentGrades[%d] ", i );
112
113       // output grades for one student
114       for ( j = 0; j < tests; ++j ) {
115          printf( "%-5d", grades[ i ][ j ] );
116       } // end inner for
117    } // end outer for
118 } // end function printArray
```

```
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. 6.22 | Double-subscripted array manipulations. (Part 3 of 3.)

Functions `minimum`, `maximum` and `printArray` each receive three arguments—the `studentGrades` array (called `grades` in each function), the number of students (rows of the array) and the number of exams (columns of 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 ( 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 ];
      } // end if
   } // end inner for
} // end outer for
```

The outer `for` statement begins by setting `i` (i.e., the row subscript) to `0` so the elements of that row (i.e., the grades of the first student) can be compared to variable `lowGrade` in the body of the inner `for` statement. The inner `for` statement loops through the four grades of a particular row and compares each grade to `lowGrade`. If a grade is less than `lowGrade`, `lowGrade` is set to that grade. The outer `for` statement then increments the row subscript to `1`. The elements of that row are compared to variable `lowGrade`. The outer `for` statement then increments the row subscript to `2`. The elements of that row are compared to variable `lowGrade`. When execution of the *nested* statement is complete, `lowGrade` contains the smallest grade in the double-subscripted array. Function `maximum` works similarly to function `minimum`.