Chapter 6
Arrays

C How to Program, 8/e, GE

Objectives

In this chapter, you'll:

- Use the array data structure to represent lists and tables of values.
- Define an array, initialize an array and refer to individual elements of an array.
- Define symbolic constants.
- Pass arrays to functions.
- Use arrays to store, sort and search lists and tables of values.
- 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.

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.

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 ([ ]).

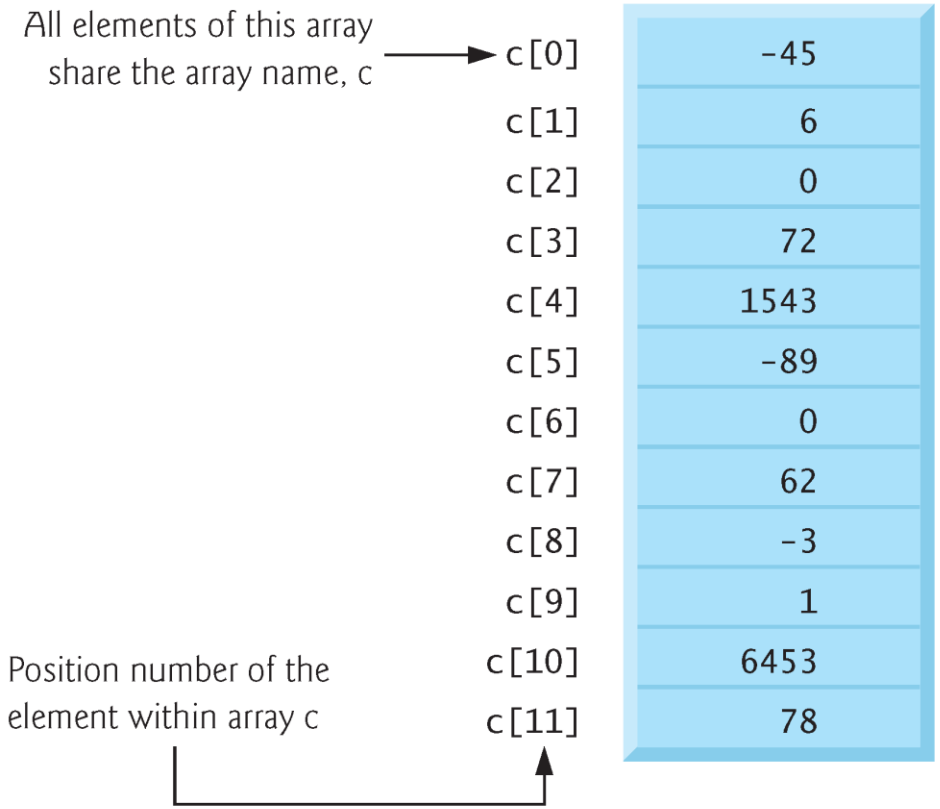

Fig. 6.1 | 12-element array.

6.2 Arrays (Cont.)

- The first element in every array is the zeroth element.
- An array name, like other identifiers, can contain only letters, digits and underscores and cannot begin with a digit.
- The position number within square brackets is called an index or subscript.
- An index must be an integer or an integer expression.

6.2 Arrays (Cont.)

- For example, if a = 5 and b = 6, then the statement
  - `c[a + b] += 2;`
- adds 2 to array element `c[11]`.
- An indexed array name is an *lvalue*—it can be used on the left side of an assignment.

6.2 Arrays (Cont.)

- 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 7 of array `c` by 2 and assign the result to the variable `x`, write

6.2 Arrays (Cont.)

- The brackets used to enclose the index 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 indices in the range 0-11.
    - `int c[12];`

6.3 Defining Arrays (Cont.)

- The definition
  - `int b[100], x[27];`

  reserves 100 elements for integer array b and 27 elements for integer array x.
- These arrays have indices in the ranges 0–99 and 0–26, respectively.
- Arrays may contain other data types.
- 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

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

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

```c
// Fig. 6.3: fig06_03.c
// Initializing the elements of an array to zeros.
#include <stdio.h>

// function main begins program execution
int main(void)
{
   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
   }

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

   // output contents of array n in tabular format
   for (size_t i = 0; i < 5; ++i) {
      printf("%7u%13d\n", i, n[i]);
   }
}
```

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
```

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

6.4 Array Examples (Cont.)

- Notice that the variable `i` is declared to be of type `size_t`, 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 indices.
- 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.]

6.4 Array Examples (Cont.)

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 five values 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[5] = {32, 27, 64, 18, 95};
10
11    printf("%s%13s\n", "Element", "Value");
12
13    // output contents of array in tabular format
14    for (size_t i = 0; i < 5; ++i) {
15       printf("%7u%13d\n", i, n[i]);
16    }
17 }
```

Fig. 6.4 | Initializing the elements of an array with an initializer list. (Part 1 of 2.)

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

Fig. 6.4 | Initializing the elements of an array with an initializer list. (Part 2 of 2.)

6.4 Array Examples (Cont.)

- 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:

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

- 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.

6.4 Array Examples (Cont.)

- 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.

6.4 Array Examples (Cont.)

- 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

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

6.4 Array Examples (Cont.)

- 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.

6.4 Array Examples (Cont.)

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.

```c
// Fig. 6.5: fig06_05.c
// Initializing the elements of array s to the even integers from 2 to 10.
#include <stdio.h>
#define SIZE 5 // maximum size of array

// function main begins program execution
int main(void)
{
   // symbolic constant SIZE can be used to specify array size
   int s[SIZE]; // array s has SIZE elements

   for (size_t j = 0; j < SIZE; ++j) { // set the values
      s[j] = 2 + 2 * j;
   }

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

   // output contents of array s in tabular format
   for (size_t j = 0; j < SIZE; ++j) {
      printf("%7u%13d\n", j, s[j]);
   }
}
```

Fig. 6.5 | Initializing the elements of array `s` to the even integers from 2 to 10. (Part 1 of 2.)

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

Fig. 6.5 | Initializing the elements of array `s` to the even integers from 2 to 10. (Part 2 of 2.)

6.4 Array Examples (Cont.)

- The `#define` preprocessor directive is introduced in this program.
- **`#define SIZE 5`**
  - defines a symbolic constant `SIZE` whose value is `5`.
- A symbolic constant is an identifier that's replaced with replacement text by the C preprocessor before the program is compiled.

6.4 Array Examples (Cont.)

- When the program is preprocessed, all occurrences of the symbolic constant `SIZE` are replaced with the replacement text `5`.
- Using symbolic constants to specify array sizes makes programs more modifiable.
- In Fig. 6.5, we could have the first `for` loop fill a 1000-element array by simply changing the value of `SIZE` in the `#define` directive from `5` to `1000`.
- If the symbolic constant `SIZE` had not been used, we'd have to change the program in *three* separate places.

Common Programming Error 6.3

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

6.4 Array Examples (Cont.)

- If the `#define` preprocessor directive is terminated with a semicolon, the preprocessor replaces all occurrences of the symbolic constant `SIZE` in the program with the text `5;`.

Software Engineering Observation 6.1

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

Common Programming Error 6.4

Assigning a value to a symbolic constant in an executable statement is a syntax error. 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.

6.4 Array Examples (Cont.)

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 does the totaling.

```c
// Fig. 6.6: fig06_06.c
// Computing the sum of the elements of an array.
#include <stdio.h>
#define SIZE 12

// function main begins program execution
int main(void)
{
   // use an initializer list to initialize the array
   int a[SIZE] = {1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45};
   int total = 0; // sum of array

   // sum contents of array a
   for (size_t i = 0; i < SIZE; ++i) {
      total += a[i];
   }

   printf("Total of array element values is %d\n", total);
}
```

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

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

```c
// Fig. 6.7: fig06_07.c
// Analyzing a student poll.
#include <stdio.h>
#define RESPONSES_SIZE 40 // define array sizes
#define FREQUENCY_SIZE 11

// function main begins program execution
int main(void)
{
   // initialize frequency counters to 0
   int frequency[FREQUENCY_SIZE] = {0};

   // place the survey responses in the responses array
   int responses[RESPONSES_SIZE] = {1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
      1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
      5, 6, 7, 5, 6, 4, 8, 6, 8, 10};

   // for each answer, select value of an element of array responses
   // and use that value as an index in array frequency to
   // determine element to increment
   for (size_t answer = 0; answer < RESPONSES_SIZE; ++answer) {
      ++frequency[responses[answer]];
   }

```

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

```
25      // display results
26      printf("%s%17s\n", "Rating", "Frequency");
27
28      // output the frequencies in a tabular format
29      for (size_t rating = 1; rating < FREQUENCY_SIZE; ++rating) {
30         printf("%6d%17d\n", rating, frequency[rating]);
31      }
32   }
```

```
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.)

6.4 Array Examples (Cont.)

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).

6.4 Array Examples (Cont.)

- The array `responses` is a 40-element array of the students' responses.
- We use an 11-element array `frequency` 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 index 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.

6.4 Array Examples (Cont.)

- The `for` loop 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
    - `++frequency[responses[answer]];`

  which increments the appropriate `frequency` counter depending on the value of `responses[answer]`.

6.4 Array Examples (Cont.)

- 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.

6.4 Array Examples (Cont.)

- 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.

6.4 Array Examples (Cont.)

- *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 index 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-continuation 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.

6.4 Array Examples (Cont.)

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 draws the bars.
- Note the use of `puts("")` to end each histogram bar.

```c
// Fig. 6.8: fig06_08.c
// Displaying a histogram.
#include <stdio.h>
#define SIZE 5

// function main begins program execution
int main(void)
{
   // use initializer list to initialize array n
   int n[SIZE] = {19, 3, 15, 7, 11};

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

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

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

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

Fig. 6.8 | Displaying a histogram. (Part 1 of 2.)

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

Fig. 6.8 | Displaying a histogram. (Part 2 of 2.)

6.4 Array Examples (Cont.)

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

- Roll a single six-sided die 60,000,000 times to test whether the random number generator actually produces random numbers.
- An array version of this program is shown in Fig. 6.9.

```c
// Fig. 6.9: fig06_09.c
// Roll a six-sided die 60,000,000 times
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 7

// function main begins program execution
int main(void)
{
   unsigned int frequency[SIZE] = {0}; // clear counts

   srand(time(NULL)); // seed random number generator

   // roll die 60,000,000 times
   for (unsigned int roll = 1; roll <= 60000000; ++roll) {
      size_t face = 1 + rand() % 6;
      ++frequency[face]; // replaces entire switch of Fig. 5.12
   }

```

Fig. 6.9 | Roll a six-sided die 60,000,000 times. (Part 1 of 2.)

```
21      printf("%s%17s\n", "Face", "Frequency");
22
23      // output frequency elements 1-6 in tabular format
24      for (size_t face = 1; face < SIZE; ++face) {
25         printf("%4d%17d\n", face, frequency[face]);
26      }
27   }
```

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

Fig. 6.9 | Roll a six-sided die 60,000,000 times. (Part 2 of 2.)

6.5 Using Character Arrays to Store and Manipulate Strings

- 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.
- 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"`.

6.5 Using Character Arrays to Store and Manipulate Strings

- 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.

6.5 Using Character Arrays to Store and Manipulate Strings

- 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 index 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`.

6.5 Using Character Arrays to Store and Manipulate Strings

- 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.

6.5 Using Character Arrays to Store and Manipulate Strings

- 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 `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 vulerability.
- 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`.

6.5 Using Character Arrays to Store and Manipulate Strings

- 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.

6.5 Using Character Arrays to Store and Manipulate Strings

- 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.

6.5 Using Character Arrays to Store and Manipulate Strings

- 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.

```c
// Fig. 6.10: fig06_10.c
// Treating character arrays as strings.
#include <stdio.h>
#define SIZE 20

// function main begins program execution
int main(void)
{
   char string1[SIZE]; // reserves 20 characters
   char string2[] = "string literal"; // reserves 15 characters

   // read string from user into array string1
   printf("%s", "Enter a string (no longer than 19 characters): ");
   scanf("%19s", string1); // input no more than 19 characters

   // output strings
   printf("string1 is: %s\nstring2 is: %s\n"
          "string1 with spaces between characters is:\n",
          string1, string2);
```

Fig. 6.10 | Treating character arrays as strings. (Part 1 of 2.)

```
20
21      // output characters until null character is reached
22      for (size_t i = 0; i < SIZE && string1[i] != '\0'; ++i) {
23         printf("%c ", string1[i]);
24      }
25
26      puts("");
27   }
```

```
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. (Part 2 of 2.)

6.6 Static Local Arrays and Automatic Local Arrays

- 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.*

6.6 Static Local Arrays and Automatic Local 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.
- Figure 6.11 demonstrates function `staticArrayInit` with a local `static` array and function `automaticArrayInit` with a local automatic array.
- Function `staticArrayInit` is called twice.
- The local `static` array in the function is initialized to zero before program startup.
- The function prints the array, adds 5 to each element and prints the array again.

6.6 Static Local Arrays and Automatic Local Arrays

- 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.
- The elements of the automatic local array in the function are initialized with the values 1, 2 and 3.
- 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.*

```c
// Fig. 6.11: fig06_11.c
// Static arrays are initialized to zero if not explicitly initialized.
#include <stdio.h>

void staticArrayInit(void); // function prototype
void automaticArrayInit(void); // function prototype

// function main begins program execution
int main(void)
{
   puts("First call to each function:");
   staticArrayInit();
   automaticArrayInit();

   puts("\n\nSecond call to each function:");
   staticArrayInit();
   automaticArrayInit();
}
```

Fig. 6.11 | Static arrays are initialized to zero if not explicitly initialized. (Part 1 of 4.)

```
20  // function to demonstrate a static local array
21  void staticArrayInit(void)
22  {
23     // initializes elements to 0 before the function is called
24     static int array1[3];
25
26     puts("\nValues on entering staticArrayInit:");
27
28     // output contents of array1
29     for (size_t i = 0; i <= 2; ++i) {
30        printf("array1[%u] = %d  ", i, array1[i]);
31     }
32
33     puts("\nValues on exiting staticArrayInit:");
34
35     // modify and output contents of array1
36     for (size_t i = 0; i <= 2; ++i) {
37        printf("array1[%u] = %d  ", i, array1[i] += 5);
38     }
39  }
40
```

Fig. 6.11 | Static arrays are initialized to zero if not explicitly initialized. (Part 2 of 4.)

```
41  // function to demonstrate an automatic local array
42  void automaticArrayInit(void)
43  {
44     // initializes elements each time function is called
45     int array2[3] = {1, 2, 3};
46
47     puts("\n\nValues on entering automaticArrayInit:");
48
49     // output contents of array2
50     for (size_t i = 0; i <= 2; ++i) {
51        printf("array2[%u] = %d  ", i, array2[i]);
52     }
53
54     puts("\nValues on exiting automaticArrayInit:");
55
56     // modify and output contents of array2
57     for (size_t i = 0; i <= 2; ++i) {
58        printf("array2[%u] = %d  ", i, array2[i] += 5);
59     }
60  }
```

Fig. 6.11 | Static arrays are initialized to zero if not explicitly initialized. (Part 3 of 4.)

```
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. 6.11 | Static arrays are initialized to zero if not explicitly initialized. (Part 4 of 4.)

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`.

6.7 Passing Arrays to Functions (Cont.)

- 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.

6.7 Passing Arrays to Functions (Cont.)

- 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.

6.7 Passing Arrays to Functions (Cont.)

- 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 numbers 10–15).
- Appendix C, Number Systems, 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`.

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.*

```c
// Fig. 6.12: fig06_12.c
// Array name is the same as the address of the array's first element.
#include <stdio.h>

// function main begins program execution
int main(void)
{
   char array[5]; // define an array of size 5

   printf("    array = %p\n&array[0] = %p\n   &array = %p\n",
      array, &array[0], &array);
}
```

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

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 placing it in a `struct` *as we explain in Chapter 10, C Structures, Unions, Bit Manipulation and Enumerations).*

6.7 Passing Arrays to Functions (Cont.)

- 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 indexed 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.

6.7 Passing Arrays to Functions (Cont.)

- 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.

6.7 Passing Arrays to Functions (Cont.)

- 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).

6.7 Passing Arrays to Functions (Cont.)

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.
- Next, `a` and its size are passed to function `modifyArray`, where each of `a`'s elements is multiplied by 2.
- Then `a` is reprinted in `main`.

6.7 Passing Arrays to Functions (Cont.)

- As the output shows, the elements of a are indeed modified by `modifyArray`.
- Now the program prints the value of `a[3]` and passes it to function `modifyElement`.
- Function `modifyElement` multiplies its argument by 2 and prints the new value.
- When `a[3]` is reprinted in `main`, it has not been modified, because individual array elements are passed by value.

6.7 Passing Arrays to Functions (Cont.)

- 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.

```c
// Fig. 6.13: fig06_13.c
// Passing arrays and individual array elements to functions.
#include <stdio.h>
#define SIZE 5

// function prototypes
void modifyArray(int b[], size_t size);
void modifyElement(int e);

// function main begins program execution
int main(void)
{
   int a[SIZE] = {0, 1, 2, 3, 4}; // initialize array a

   puts("Effects of passing entire array by reference:\n\nThe "
      "values of the original array are:");

   // output original array
   for (size_t i = 0; i < SIZE; ++i) {
      printf("%3d", a[i]);
   }

   puts(""); // outputs a newline

```

Fig. 6.13 | Passing arrays and individual array elements to functions. (Part 1 of 4.)

```
25      modifyArray(a, SIZE); // pass array a to modifyArray by reference
26      puts("The values of the modified array are:");
27
28      // output modified array
29      for (size_t i = 0; i < SIZE; ++i) {
30         printf("%3d", a[i]);
31      }
32
33      // output value of a[3]
34      printf("\n\n\nEffects of passing array element "
35         "by value:\n\nThe value of a[3] is %d\n", a[3]);
36
37      modifyElement(a[3]); // pass array element a[3] by value
38
39      // output value of a[3]
40      printf("The value of a[3] is %d\n", a[3]);
41   }
42
```

Fig. 6.13 | Passing arrays and individual array elements to functions. (Part 2 of 4.)

```
43  // in function modifyArray, "b" points to the original array "a"
44  // in memory
45  void modifyArray(int b[], size_t size)
46  {
47     // multiply each array element by 2
48     for (size_t j = 0; j < size; ++j) {
49        b[j] *= 2; // actually modifies original array
50     }
51  }
52
53  // in function modifyElement, "e" is a local copy of array element
54  // a[3] passed from main
55  void modifyElement(int e)
56  {
57     // multiply parameter by 2
58     printf("Value in modifyElement is %d\n", e *= 2);
59  }
```

Fig. 6.13 | Passing arrays and individual array elements to functions. (Part 3 of 4.)

```
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 4 of 4.)

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.*

6.7 Passing Arrays to Functions (Cont.)

Using the `const` Qualifier with Array Parameters

- Figure 6.14 demonstrates the `const` qualifier.
- Function `tryToModifyArray` 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.`" .

```
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  {
5     b[0] /= 2; // error
6     b[1] /= 2; // error
7     b[2] /= 2; // error
8  }
```

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

6.8 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.
- 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.

6.8 Sorting Arrays

- Figure 6.15 sorts the values in the elements of the 10-element array a 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.

6.8 Sorting Arrays

- 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.

```c
// Fig. 6.15: fig06_15.c
// Sorting an array's values into ascending order.
#include <stdio.h>
#define SIZE 10

// function main begins program execution
int main(void)
{
   // initialize a
   int a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};

   puts("Data items in original order");

   // output original array
   for (size_t i = 0; i < SIZE; ++i) {
      printf("%4d", a[i]);
   }

```

Fig. 6.15 | Sorting an array's values into ascending order. (Part 1 of 3.)

```
19      // bubble sort
20      // loop to control number of passes
21      for (unsigned int pass = 1; pass < SIZE; ++pass) {
22
23         // loop to control number of comparisons per pass
24         for (size_t i = 0; i < SIZE - 1; ++i) {
25
26            // compare adjacent elements and swap them if first
27            // element is greater than second element
28            if (a[i] > a[i + 1]) {
29               int hold = a[i];
30               a[i] = a[i + 1];
31               a[i + 1] = hold;
32            }
33         }
34      }
35
```

Fig. 6.15 | Sorting an array's values into ascending order. (Part 2 of 3.)

```
36      puts("\nData items in ascending order");
37
38      // output sorted array
39      for (size_t i = 0; i < SIZE; ++i) {
40         printf("%4d", a[i]);
41      }
42
43      puts("");
44   }
```

```
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 3 of 3.)

6.8 Sorting Arrays

- 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]`.

6.8 Sorting Arrays

- 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.

6.8 Sorting Arrays

- 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];
    ```

6.8 Sorting Arrays

- 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.

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

- 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.

```c
// Fig. 6.16: fig06_16.c
// Survey data analysis with arrays:
// computing the mean, median and mode of the data.
#include <stdio.h>
#define SIZE 99

// function prototypes
void mean(const unsigned int answer[]);
void median(unsigned int answer[]);
void mode(unsigned int freq[], unsigned const int answer[]) ;
void bubbleSort(int a[]);
void printArray(unsigned const int a[]);

// function main begins program execution
int main(void)
{
   unsigned int frequency[10] = {0}; // initialize array frequency

```

Fig. 6.16 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 1 of 8.)

```
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   }
37
```

Fig. 6.16 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 2 of 8.)

```
38  // calculate average of all response values
39  void mean(const unsigned int answer[])
40  {
41     printf("%s\n%s\n%s\n", "********", "  Mean", "********");
42
43     unsigned int total = 0; // variable to hold sum of array elements
44
45     // total response values
46     for (size_t j = 0; j < SIZE; ++j) {
47        total += answer[j];
48     }
49
50     printf("The mean is the average value of the data\n"
51            "items. The mean is equal to the total of\n"
52            "all the data items divided by the number\n"
53            "of data items (%u). The mean value for\n"
54            "this run is: %u / %u = %.4f\n\n",
55            SIZE, total, SIZE, (double) total / SIZE);
56  }
57
```

Fig. 6.16 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 3 of 8.)

```
58  // sort array and determine median element's value
59  void median(unsigned int answer[])
60  {
61     printf("\n%s\n%s\n%s\n%s",
62             "********", " Median", "********",
63             "The unsorted array of responses is");
64
65     printArray(answer); // output unsorted array
66
67     bubbleSort(answer); // sort array
68
69     printf("%s", "\n\nThe sorted array is");
70     printArray(answer); // output sorted array
71
72     // display median element
73     printf("\n\nThe median is element %u of\n"
74            "the sorted %u element array.\n"
75            "For this run the median is %u\n\n",
76            SIZE / 2, SIZE, answer[SIZE / 2]);
77  }
78
```

Fig. 6.16 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 4 of 8.)

```
79  // determine most frequent response
80  void mode(unsigned int freq[], const unsigned int answer[])
81  {
82     printf("\n%s\n%s\n%s\n","********", "  Mode", "********");
83
84     // initialize frequencies to 0
85     for (size_t rating = 1; rating <= 9; ++rating) {
86        freq[rating] = 0;
87     }
88
89     // summarize frequencies
90     for (size_t j = 0; j < SIZE; ++j) {
91        ++freq[answer[j]];
92     }
93
94     // output headers for result columns
95     printf("%s%11s%19s\n\n%54s\n%54s\n\n",
96            "Response", "Frequency", "Histogram",
97            "1    1    2    2", "5    0    5    0    5");
98
99     // output results
100    unsigned int largest = 0; // represents largest frequency
101    unsigned int modeValue = 0; // represents most frequent response
```

Fig. 6.16 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 5 of 8.)

```
102
103     for (rating = 1; rating <= 9; ++rating) {
104        printf("%8u%11u           ", rating, freq[rating]);
105
106        // keep track of mode value and largest frequency value
107        if (freq[rating] > largest) {
108           largest = freq[rating];
109           modeValue = rating;
110        }
111
112        // output histogram bar representing frequency value
113        for (unsigned int h = 1; h <= freq[rating]; ++h) {
114           printf("%s", "*");
115        }
116
117        puts(""); // being new line of output
118     }
119
120     // display the mode value
121     printf("\nThe mode is the most frequent value.\n"
122            "For this run the mode is %u which occurred"
123            " %u times.\n", modeValue, largest);
124  }
```

Fig. 6.16 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 6 of 8.)

```
125
126 // function that sorts an array with bubble sort algorithm
127 void bubbleSort(unsigned int a[])
128 {
129    // loop to control number of passes
130    for (unsigned int pass = 1; pass < SIZE; ++pass) {
131
132       // loop to control number of comparisons per pass
133       for (size_t j = 0; j < SIZE - 1; ++j) {
134
135          // swap elements if out of order
136          if (a[j] > a[j + 1]) {
137             unsigned int hold = a[j];
138             a[j] = a[j + 1];
139             a[j + 1] = hold;
140          }
141       }
142    }
143 }
144
```

Fig. 6.16 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 7 of 8.)

```
145 // output array contents (20 values per row)
146 void printArray(const unsigned int a[])
147 {
148    // output array contents
149    for (size_t j = 0; j < SIZE; ++j) {
150
151       if (j % 20 == 0) { // begin new line every 20 values
152          puts("");
153       }
154
155       printf("%2u", a[j]);
156    }
157 }
```

Fig. 6.16 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 8 of 8.)

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

Fig. 6.17 | Sample run for the survey data analysis program. (Part 1 of 3.)

```
********
 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

The sorted array is
 1 2 2 2 3 3 3 3 4 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 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 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
```

Fig. 6.17 | Sample run for the survey data analysis program. (Part 2 of 3.)

```
********
  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 3 of 3.)

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

Mean

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

Median

- The median is the "*middle* value."
- Function `median` determines the median by calling function `bubbleSort` to sort the array of responses into ascending order, then picking `answer[SIZE / 2]` (the middle element) of the sorted array.

6.9 Case Study: Computing Mean, Median and Mode Using Arrays (Cont.)

- 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` is called to output the `response` array.

6.9 Case Study: Computing Mean, Median and Mode Using Arrays (Cont.)

Mode

- The *mode* is the *value that occurs most frequently* among the 99 responses.
- Function `mode` 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 7.14).
- Function `mode` also produces a histogram to aid in determining the mode graphically.

6.10 Searching 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.

6.10 Searching Arrays (Cont.)

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.

```c
// Fig. 6.18: fig06_18.c
// Linear search of an array.
#include <stdio.h>
#define SIZE 100

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

// function main begins program execution
int main(void)
{
   int a[SIZE]; // 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; // value to locate in array a
   scanf("%d", &searchKey);

```

Fig. 6.18 | Linear search of an array. (Part 1 of 3.)

```
23      // attempt to locate searchKey in array a
24      size_t index = linearSearch(a, searchKey, SIZE);
25
26      // display results
27      if (index != -1) {
28         printf("Found value at index %d\n", index);
29      }
30      else {
31         puts("Value not found");
32      }
33   }
34
```

Fig. 6.18 | Linear search of an array. (Part 2 of 3.)

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

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

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

Fig. 6.18 | Linear search of an array. (Part 3 of 3.)

6.10 Searching Arrays (Cont.)

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.

6.10 Searching Arrays (Cont.)

- If they're equal, the search key is found and the index 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* of the array 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.

6.10 Searching Arrays (Cont.)

- 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 is 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.

6.10 Searching Arrays (Cont.)

- 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!

6.10 Searching Arrays (Cont.)

- 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`
- The function receives four arguments—an integer array `b` to be searched, an integer `searchKey`, the `low` array index and the `high` array index (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` index or `high` index is modified so that a smaller subarray can be searched.

6.10 Searching Arrays (Cont.)

- If the search key is *less than* the middle element, the `high` index 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` index 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.

6.10 Searching Arrays (Cont.)

- The program uses function `printHeader` to output the array indices and function `printRow` 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.

```c
// Fig. 6.19: fig06_19.c
// Binary search of a sorted array.
#include <stdio.h>
#define SIZE 15

// function prototypes
size_t binarySearch(const int b[], int searchKey, size_t low, size_t high);
void printHeader(void);
void printRow(const int b[], size_t low, size_t mid, size_t high);

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

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

   printf("%s", "Enter a number between 0 and 28: ");
   int key; // value to locate in array a
   scanf("%d", &key);

```

Fig. 6.19 | Binary search of a sorted array. (Part 1 of 7.)

```
25      printHeader();
26
27      // search for key in array a
28      size_t result = binarySearch(a, key, 0, SIZE - 1);
29
30      // display results
31      if (result != -1) {
32         printf("\n%d found at index %d\n", key, result);
33      }
34      else {
35         printf("\n%d not found\n", key);
36      }
37   }
38
39   // function to perform binary search of an array
40   size_t binarySearch(const int b[], int searchKey, size_t low, size_t high)
41   {
42      // loop until low index is greater than high index
43      while (low <= high) {
44
45         // determine middle element of subarray being searched
46         size_t middle = (low + high) / 2;
47
```

Fig. 6.19 | Binary search of a sorted array. (Part 2 of 7.)

```
48          // display subarray used in this loop iteration
49          printRow(b, low, middle, high);
50
51          // if searchKey matched middle element, return middle
52          if (searchKey == b[middle]) {
53             return middle;
54          }
55
56          // if searchKey is less than middle element, set new high
57          else if (searchKey < b[middle]) {
58             high = middle - 1; // search low end of array
59          }  if
60
61          // if searchKey is greater than middle element, set new low
62          else {
63             low = middle + 1; // search high end of array
64          }
65       } // end while
66
67       return -1; // searchKey not found
68    }
69
```

Fig. 6.19 | Binary search of a sorted array. (Part 3 of 7.)

```
70  // Print a header for the output
71  void printHeader(void)
72  {
73     puts("\nIndices:");
74
75     // output column head
76     for (unsigned int i = 0; i < SIZE; ++i) {
77        printf("%3u ", i);
78     }
79
80     puts(""); // start new line of output
81
82     // output line of - characters
83     for (unsigned int i = 1; i <= 4 * SIZE; ++i) {
84        printf("%s", "-");
85     }
86
87     puts(""); // start new line of output
88  }
89
```

Fig. 6.19 | Binary search of a sorted array. (Part 4 of 7.)

```
90  // Print one row of output showing the current
91  // part of the array being processed.
92  void printRow(const int b[], size_t low, size_t mid, size_t high)
93  {
94     // loop through entire array
95     for (size_t i = 0; i < SIZE; ++i) {
96
97        // display spaces if outside current subarray range
98        if (i < low || i > high) {
99           printf("%s", "    ");
100       }
101       else if (i == mid) { // display middle element
102          printf("%3d*", b[i]); // mark middle value
103       }
104       else { // display other elements in subarray
105          printf("%3d ", b[i]);
106       }
107    }
108
109    puts(""); // start new line of output
110 }
```

Fig. 6.19 | Binary search of a sorted array. (Part 5 of 7.)

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

Indices:
  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 6 of 7.)

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

Indices:
  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 index 4
```

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

Indices:
  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 index 3
```

Fig. 6.19 | Binary search of a sorted array. (Part 7 of 7.)

6.11 Multidimensional Arrays

- Arrays in C can have multiple indices.
- 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 must specify two indices: 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 indices to identify a particular element are called two-dimensional arrays.

6.11 Multidimensional Arrays (Cont.)

- Multidimensional arrays can have more than two indices.
- Figure 6.20 illustrates a two-dimensional 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

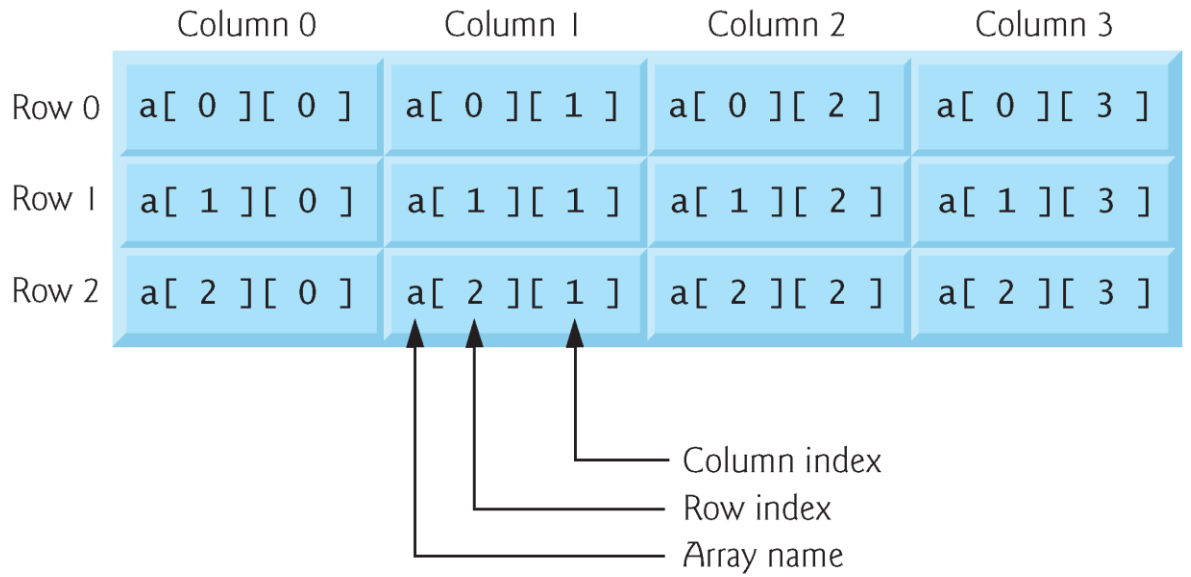

Fig. 6.20 | Two-dimensional array with three rows and four columns.

6.11 Multidimensional Arrays (Cont.)

- 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 indices that uniquely identify each element in `a`.
- The names of the elements in row `0` all have a first index of `0`; the names of the elements in column 3 all have a second index of 3.

Common Programming Error 6.7

Referencing a two-dimensional 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.*

6.11 Multidimensional Arrays (Cont.)

- A multidimensional array can be initialized when it's defined, much like a one-dimensional array.
- For example, a two-dimensional 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.

6.11 Multidimensional Arrays (Cont.)

- *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 two-dimensional arrays.

```c
// Fig. 6.21: fig06_21.c
// Initializing multidimensional arrays.
#include <stdio.h>

void printArray(int a[][3]); // function prototype

// function main begins program execution
int main(void)
{
   int array1[2][3] = {{1, 2, 3}, {4, 5, 6}};
   puts("Values in array1 by row are:");
   printArray(array1);

   int array2[2][3] = {1, 2, 3, 4, 5};
   puts("Values in array2 by row are:");
   printArray(array2);

   int array3[2][3] = {{1, 2}, {4}};
   puts("Values in array3 by row are:");
   printArray(array3);
}

```

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

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

```
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.)

6.11 Multidimensional Arrays (Cont.)

- The program defines three arrays of two rows and three columns (six elements each).
- The definition of `array1` 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.

6.11 Multidimensional Arrays (Cont.)

- 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` 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` provides three initializers in two sublists.

6.11 Multidimensional Arrays (Cont.)

- 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` to output each array's elements.
- The function definition specifies the array parameter as `const int a[][3]`.
- When we receive a one-dimensional array as a parameter, the array brackets are *empty* in the function's parameter list.

6.11 Multidimensional Arrays (Cont.)

- The first index of a multidimensional array is not required either, but all subsequent indices are required.
- The compiler uses these indices to determine the locations in memory of elements in multidimensional arrays.
- All array elements are stored consecutively in memory regardless of the number of indices.
- In a two-dimensional array, the first row is stored in memory followed by the second row.
- Providing the index values in a parameter declaration enables the compiler to tell the function how to locate an element in the array.

6.11 Multidimensional Arrays (Cont.)

- 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.
- 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.

6.11 Multidimensional Arrays (Cont.)

- Many common array manipulations use `for` iteration statements.
- For example, the following statement sets all the elements in row 2 of array `a` in Fig. 6.20 to zero:
  - ```
    for (size_t column = 0; column <= 3; ++column) {
       a[2][column] = 0;
    }
    ```
- We specified row 2, so the first index is always 2.
    ```

6.11 Multidimensional Arrays (Cont.)

- The `loop` varies only the second index.
- 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 Multidimensional Arrays (Cont.)

- The following nested `for` statement determines the total of all the elements in array `a`.
  - `total = 0;`
  - ```
    for (size_t row = 0; row <= 2; ++row) {
       for (size_t column = 0; column <= 3; ++column) {
          total += a[row][column];
       }
    }
    ```
- The `for` statement totals the elements of the array one row at a time.
    ```

6.11 Multidimensional Arrays (Cont.)

- The outer `for` statement begins by setting `row` (i.e., the row index) 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`.

6.11 Multidimensional Arrays (Cont.)

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` determines the lowest grade of any student for the semester.

6.11 Multidimensional Arrays (Cont.)

- Function `maximum` determines the highest grade of any student for the semester.
- Function `average` determines a particular student's semester average.
- Function `printArray` outputs the two-dimensional array in a neat, tabular format.

```c
// Fig. 6.22: fig06_22.c
// Two-dimensional array manipulations.
#include <stdio.h>
#define STUDENTS 3
#define EXAMS 4

// function prototypes
int minimum(const int grades[][EXAMS], size_t pupils, size_t tests);
int maximum(const int grades[][EXAMS], size_t pupils, size_t tests);
double average(const int setOfGrades[], size_t tests);
void printArray(const int grades[][EXAMS], size_t pupils, size_t tests);

// function main begins program execution
int main(void)
{
   // initialize student grades for three students (rows)
   int studentGrades[STUDENTS][EXAMS] =
      { { 77, 68, 86, 73 },
        { 96, 87, 89, 78 },
        { 70, 90, 86, 81 } };

   // output array studentGrades
   puts("The array is:");
   printArray(studentGrades, STUDENTS, EXAMS);
```

Fig. 6.22 | Two-dimensional array manipulations. (Part 1 of 7.)

```
25
26      // determine smallest and largest grade values
27      printf("\n\nLowest grade: %d\nHighest grade: %d\n",
28         minimum(studentGrades, STUDENTS, EXAMS),
29         maximum(studentGrades, STUDENTS, EXAMS));
30
31      // calculate average grade for each student
32      for (size_t student = 0; student < STUDENTS; ++student) {
33         printf("The average grade for student %u is %.2f\n",
34            student, average(studentGrades[student], EXAMS));
35      }
36   }
37
```

Fig. 6.22 | Two-dimensional array manipulations. (Part 2 of 7.)

```
38  // Find the minimum grade
39  int minimum(const int grades[][EXAMS], size_t pupils, size_t tests)
40  {
41     int lowGrade = 100; // initialize to highest possible grade
42
43     // loop through rows of grades
44     for (size_t i = 0; i < pupils; ++i) {
45
46        // loop through columns of grades
47        for (size_t j = 0; j < tests; ++j) {
48
49           if (grades[i][j] < lowGrade) {
50              lowGrade = grades[i][j];
51           }
52        }
53     }
54
55     return lowGrade; // return minimum grade
56  }
57
```

Fig. 6.22 | Two-dimensional array manipulations. (Part 3 of 7.)

```
58  // Find the maximum grade
59  int maximum(const int grades[][EXAMS], size_t pupils, size_t tests)
60  {
61     int highGrade = 0; // initialize to lowest possible grade
62
63     // loop through rows of grades
64     for (size_t i = 0; i < pupils; ++i) {
65
66        // loop through columns of grades
67        for (size_t j = 0; j < tests; ++j) {
68
69           if (grades[i][j] > highGrade) {
70              highGrade = grades[i][j];
71           }
72        }
73     }
74
75     return highGrade; // return maximum grade
76  }
77
```

Fig. 6.22 | Two-dimensional array manipulations. (Part 4 of 7.)

```
78  // Determine the average grade for a particular student
79  double average(const int setOfGrades[], size_t tests)
80  {
81     int total = 0; // sum of test grades
82
83     // total all grades for one student
84     for (size_t i = 0; i < tests; ++i) {
85        total += setOfGrades[i];
86     }
87
88     return (double) total / tests; // average
89  }
90
```

Fig. 6.22 | Two-dimensional array manipulations. (Part 5 of 7.)

```
91  // Print the array
92  void printArray(const int grades[][EXAMS], size_t pupils, size_t tests)
93  {
94     // output column heads
95     printf("%s", "                  [0]  [1]  [2]  [3]");
96
97     // output grades in tabular format
98     for (size_t i = 0; i < pupils; ++i) {
99
100       // output label for row
101       printf("\nstudentGrades[%u] ", i);
102
103       // output grades for one student
104       for (size_t j = 0; j < tests; ++j) {
105          printf("%-5d", grades[i][j]);
106       }
107    }
108 }
```

Fig. 6.22 | Two-dimensional array manipulations. (Part 6 of 7.)

```
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 | Two-dimensional array manipulations. (Part 7 of 7.)

6.11 Multidimensional Arrays (Cont.)

- 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.

6.11 Multidimensional Arrays (Cont.)

- The following nested `for` statement is from the function `minimum` definition:
  - ```
    // loop through rows of grades
    for (size_t i = 0; i < pupils; ++i) {
       // loop through columns of grades
       for (size_t j = 0; j < tests; ++j) {
          if (grades[i][j] < lowGrade) {
             lowGrade = grades[i][j];
          }
       }
    }
    ```
    ```

6.11 Multidimensional Arrays (Cont.)

- The outer `for` statement begins by setting `i` (i.e., the row index) to `0` so that 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 index to `1`.
- The elements of that row are compared to variable `lowGrade`.

6.11 Multidimensional Arrays (Cont.)

- The outer `for` statement then increments the row index 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 two-dimensional array.
- Function `maximum` works similarly to function `minimum`.
- Function `average` takes two arguments—a one-dimensional array of test results for a particular student called `setOfGrades` and the number of test results in the array.

6.11 Multidimensional Arrays (Cont.)

- When `average` is called, the first argument `studentGrades[student]` is passed.
- This causes the address of one row of the two-dimensional array to be passed to `average`.
- 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 that the name of a one-dimensional array is the address of the 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.

6.12 Variable-Length Arrays

- In early versions of C, all arrays had constant size.
- But what if you don't know an array's size at compilation time?
- To handle this, you'd have to use dynamic memory allocation with `malloc` and related functions.
- The C standard allows you to handle arrays of unknown size using variable-length arrays (VLAs).
- These are not arrays whose size can change—that would compromise the integrity of nearby locations in memory.

6.12 Variable-Length Arrays (Cont.)

- A variable-length array is an array whose length, or size, is defined in terms of an expression evaluated at execution time.
- The program of Fig. 6.23 declares and prints several VLAs.
- [Note: This feature is not supported in Microsoft Visual C++.]

```c
// Fig. 6.23: fig06_23.c
// Using variable-length arrays in C99
#include <stdio.h>

// function prototypes
void print1DArray(size_t size, int array[size]);
void print2DArray(int row, int col, int array[row][col]);

int main(void)
{
   printf("%s", "Enter size of a one-dimensional array: ");
   int arraySize; // size of 1-D array
   scanf("%d", &arraySize);

   int array[arraySize]; // declare 1-D variable-length array

   printf("%s", "Enter number of rows and columns in a 2-D array: ");
   int row1, col1; // number of rows and columns in a 2-D array
   scanf("%d %d", &row1, &col1);

   int array2D1[row1][col1]; // declare 2-D variable-length array

```

Fig. 6.23 | Using variable-length arrays in C99. (Part 1 of 5.)

```
23      printf("%s",
24         "Enter number of rows and columns in another 2-D array: ");
25      int row2, col2; // number of rows and columns in another 2-D array
26      scanf("%d %d", &row2, &col2);
27
28      int array2D2[row2][col2]; // declare 2-D variable-length array
29
30      // test sizeof operator on VLA
31      printf("\nsizeof(array) yields array size of %d bytes\n",
32         sizeof(array));
33
34      // assign elements of 1-D VLA
35      for (size_t i = 0; i < arraySize; ++i) {
36         array[i] = i * i;
37      }
38
39      // assign elements of first 2-D VLA
40      for (size_t i = 0; i < row1; ++i) {
41         for (size_t j = 0; j < col1; ++j) {
42            array2D1[i][j] = i + j;
43         }
44      }
45
```

Fig. 6.23 | Using variable-length arrays in C99. (Part 2 of 5.)

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

Fig. 6.23 | Using variable-length arrays in C99. (Part 3 of 5.)

```
70
71  void print2DArray(size_t row, size_t col, int array[row][col])
72  {
73     // output contents of array
74     for (size_t i = 0; i < row; ++i) {
75        for (size_t j = 0; j < col; ++j) {
76           printf("%5d", array[i][j]);
77        }
78
79        puts("");
80     }
81  }
```

Fig. 6.23 | Using variable-length arrays in C99. (Part 4 of 5.)

```
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

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

Fig. 6.23 | Using variable-length arrays in C99. (Part 5 of 5.)

6.12 Variable-Length Arrays (Cont.)

- First, we prompt the user for the desired sizes for a one-dimensional array and two two-dimensional arrays .
- Next we declare VLAs of the appropriate size.
- This is valid as long as the variables representing the array sizes are of an integral type.
- After declaring the arrays, we use the `sizeof` operator to make sure that our VLA is of the proper length.
- In early versions of C `sizeof` was always a compile-time operation, but when applied to a VLA, `sizeof` operates at runtime.

6.12 Variable-Length Arrays (Cont.)

- The output window shows that the `sizeof` operator returns a size of 24 bytes—four times that of the number we entered because the size of an `int` on our machine is 4 bytes.
- Next we assign values to the elements of our VLAs.
- We use `i < arraySize` as our loop-continuation condition when filling the one-dimensional array.
- As with fixed-length arrays, there is no protection against stepping outside the array bounds.
- Function `print1DArray` takes a one-dimensional VLA.

6.12 Variable-Length Arrays (Cont.)

- The syntax for passing VLAs as parameters to functions is the same as with a normal, fixed-length array.
- We use the variable `size` in the declaration of the array parameter, but no checking is performed other than the variable being defined and of integral type—it's purely documentation for the programmer.
- Function `print2DArray` takes a variable-length two-dimensional array and displays it to the screen.
- Recall from Section 6.9 that, all but the first index of a multidimensional array must be specified when declaring a function parameter.

6.12 Variable-Length Arrays (Cont.)

- 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 is used to convert from two-dimensional indices to offsets into the contiguous memory the array is stored in, just as with a fixed-size array.
- Changing the value of `col` inside the function will not cause any changes to the indexing, but passing an incorrect value to the function will.

6.13 Secure C Programming

Bounds Checking for Array Indices

- It's important to ensure that every index you use to access an array element is within the array's bounds—that is, greater than or equal to 0 and less than the number of array elements.
- A two-dimensional array's row and column indices must be greater than or equal to 0 and less than the numbers of rows and columns, respectively.
- Allowing programs to read from or write to array elements outside the bounds of arrays 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.

6.13 Secure C Programming (Cont.)

- Writing to an out-of-bounds element (known as a *buffer overflow*) can corrupt a program's data in memory, crash a program and allow attackers to exploit the system and execute their own code.
- As we stated in the chapter, *C provides no automatic bounds checking for arrays*, so you must provide your own.
- For techniques that help you prevent such problems, see CERT guideline ARR30-C at www.securecoding.cert.org.

6.13 Secure C Programming (Cont.)

scanf_s

- Bounds checking is also important in string processing.
- When reading a string into a `char` array, `scanf` does not 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, and eventually the program might overwrite the string's `'\0'` if it writes to those other variables.

6.13 Secure C Programming (Cont.)

- Functions determine where strings end by looking for their terminating `'\0'` character.
- For example, function `printf` outputs a string by reading characters from the beginning of the string in memory and continuing until the string's `'\0'` is encountered.
- If the `'\0'` is missing, `printf` might read far beyond the end of the string until it encounters some other `'\0'` in memory.

6.13 Secure C Programming (Cont.)

- The C standard's optional Annex K provides new, more secure, versions of many string-processing and input/output functions, including `scanf_s`—a version of `scanf` that performs additional checks to ensure that it *does not* write beyond the end of a character array used to store a string.
- 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 number of array elements.

6.13 Secure C Programming (Cont.)

- The second of these arguments is used by `scanf_s` 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.
- If the number of characters input plus the terminating null character is larger than the number of array elements, the %s conversion would fail.
- Because the preceding statement contains only one conversion specifier, `scanf_s` would return 0 indicating that no conversions were performed, and `myString` would be unaltered.
- In general, if your compiler supports the functions from the C standard's optional Annex K, you should use them.

6.13 Secure C Programming (Cont.)

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

- You might have noticed that throughout this book, we never use single-argument printfs. 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: ");
    ```

6.13 Secure C Programming (Cont.)

- 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 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.

6.13 Secure C Programming (Cont.)

- 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 www.securecoding.cert.org.

Portability Tip 6.1

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.*