



C Arrays

C How to Program, 6/e



OBJECTIVES

In this chapter, you'll learn:

- To use the array data structure to represent lists and tables of values.
- To define an array, initialize an array and refer to individual elements of an array.
- To define symbolic constants.
- To pass arrays to functions.
- To use arrays to store, sort and search lists and tables of values.
- To define and manipulate multiple-subscripted arrays.



- 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 Multiple-Subscripted Arrays



6.1 Introduction

- ▶ This chapter serves as an introduction to the important topic of 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.
- ▶ In Chapter 12, we introduce dynamic data structures such as lists, queues, stacks and trees that may grow and shrink as programs execute.



6.2 Arrays

- ▶ An array is a group of memory locations related by the fact that they all have the same name and the same type.
- ▶ To refer to a particular location or element in the array, we specify the name of the array and the **position number** of the particular element in the array.
- ▶ Figure 6.1 shows an integer array called **c**.
- ▶ This array contains 12 **elements**.
- ▶ Any one of these elements may be referred to by giving the name of the array followed by the position number of the particular element in square brackets (**[]**).



6.2 Arrays (Cont.)

- ▶ The first element in every array is the **zeroth element**.
- ▶ Thus, the first element of array `C` is referred to as `C[0]`, the second element of array `C` is referred to as `C[1]`, the seventh element of array `C` is referred to as `C[6]`, and, in general, the i th element of array `C` is referred to as `C[i - 1]`.
- ▶ Array names, like other variable names, can contain only letters, digits and underscores.
- ▶ Array names cannot begin with a digit.
- ▶ The position number contained within square brackets is more formally called a **subscript** (or **index**).
- ▶ A subscript must be an integer or an integer expression.



6.2 Arrays (Cont.)

- ▶ If a program uses an expression as a subscript, then the expression is evaluated to determine the subscript.
- ▶ 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.



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[11]`.
- ▶ The **value** stored in `c[0]` is `-45`, the value of `c[1]` is `6`, the value of `c[2]` is `0`, the value of `c[7]` is `62` and the value of `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 the seventh element of array `c` by `2` and assign the result to the variable `x`, we'd write

Name of array (note that all elements
of this array have the same name, c)

c[0]	-45
c[1]	6
c[2]	0
c[3]	72
c[4]	1543
c[5]	-89
c[6]	0
c[7]	62
c[8]	-3
c[9]	1
c[10]	6453
c[11]	78

Position number of the element within array c

Fig. 6.1 | 12-element array.



Common Programming Error 6.1

It's important to note the difference between the “seventh element of the array” and “array element seven.” Because array subscripts begin at 0, the “seventh element of the array” has a subscript of 6, while “array element seven” has a subscript of 7 and is actually the eighth element of the array. This is a source of “off-by-one” errors.



6.2 Arrays (Cont.)

- ▶ 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 following 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
[]	()					left to right	highest
++	--	!	(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 required by each array so that the computer may reserve the appropriate amount of memory.
- ▶ To tell the computer to reserve 12 elements for integer array **C**, use the definition
 - `int c[ 12 ];`



6.3 Defining Arrays (Cont.)

- ▶ The following definition
 - `int b[ 100 ], x[ 27 ];`
reserves 100 elements for integer array **b** and 27 elements for integer array **x**.
- ▶ 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

- ▶ 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.



```
1  /* Fig. 6.3: fig06_03.c
2     initializing an array */
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     int 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( "%7d%13d\n", i, n[ i ] );
21     } /* end for */
22
23     return 0; /* indicates successful termination */
24 } /* end main */
```

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



6.4 Array Examples (Cont.)

- ▶ 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 **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 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    int 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( "%7d%13d\n", i, n[ i ] );
17    } /* end for */
18
19    return 0; /* indicates successful termination */
20 } /* end main */
```

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
5	14
6	90
7	70
8	60
9	37

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:
 - `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.
- ▶ This method of initializing the array elements to 0 is performed at compile time for **static** arrays and at runtime for automatic arrays.



Common Programming Error 6.2

Forgetting to initialize the elements of an array whose elements should be initialized.



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 6.3

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



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.



6.4 Array Examples (Cont.)

- ▶ 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     Initialize 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    int j; /* counter */
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( "%7d%13d\n", j, s[ j ] );
22    } /* end for */
```

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



```
23
24     return 0; /* indicates successful termination */
25 } /* 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.)



6.4 Array Examples (Cont.)

- ▶ 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 is 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 **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 to scale the program to handle 1000 array elements.



Common Programming Error 6.4

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 in line 4 is terminated with a semicolon, all occurrences of the symbolic constant `SIZE` in the program are replaced with the text `10;` by the preprocessor.
- ▶ This may lead to syntax errors at compile time, or logic errors at execution time.



Common Programming Error 6.5

Assigning a value to a symbolic constant in an executable statement is a syntax error. A symbolic constant is not a variable. No space is reserved for it by the compiler as with variables that hold values at execution time.



Software Engineering Observation 6.1

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



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, use underscores to separate the words for readability.



6.4 Array Examples (Cont.)

- ▶ 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     Compute the sum of the elements of the array */
3  #include <stdio.h>
4  #define SIZE 12
5
6  /* function main begins program execution */
7  int main( void )
8  {
9     /* use initializer list to initialize array */
10    int a[ SIZE ] = { 1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45 };
11    int 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    return 0; /* indicates successful termination */
21 } /* end main */
```

Total of array element values is 383

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

6.4 Array Examples (Cont.)

- ▶ 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** (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.



```
1  /* Fig. 6.7: fig06_07.c
2     Student poll program */
3  #include <stdio.h>
4  #define RESPONSE_SIZE 40 /* define array sizes */
5  #define FREQUENCY_SIZE 11
6
7  /* function main begins program execution */
8  int main( void )
9  {
10     int answer; /* counter to loop through 40 responses */
11     int 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[ RESPONSE_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
```

Fig. 6.7 | Student poll analysis program. (Part I of 3.)



```
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 < RESPONSE_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
36  return 0; /* indicates successful termination */
37  } /* end main */
```

Fig. 6.7 | Student poll analysis program. (Part 2 of 3.)



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 | Student poll analysis program. (Part 3 of 3.)



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 (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]**.

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 actually 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” the end of an array without warning.
- ▶ You should ensure that all array references remain within the bounds of the array.



Common Programming Error 6.6

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 ($\text{size} - 1$). 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.



6.4 Array Examples (Cont.)

- ▶ 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 `printf( "\\n" )` to end a histogram bar (line 24).



```
1  /* Fig. 6.8: fig06_08.c
2     Histogram printing program */
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    int 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( "%7d%13d", i, n[ i ] );
19
20        for ( j = 1; j <= n[ i ]; j++ ) { /* print one bar */
21            printf( "%c", '*' );
22        } /* end inner for */
23    }
```

Fig. 6.8 | Histogram printing. (Part I of 2.)

```
24     printf( "\n" ); /* end a histogram bar */
25 } /* end outer for */
26
27 return 0; /* indicates successful termination */
28 }
```

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 | Histogram printing. (Part 2 of 2.)



6.4 Array Examples (Cont.)

- Roll a single six-sided die 6000 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 6000 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     int face; /* random die value 1 - 6 */
12     int roll; /* roll counter 1-6000 */
13     int frequency[ SIZE ] = { 0 }; /* clear counts */
14
15     srand( time( NULL ) ); /* seed random-number generator */
16
17     /* roll die 6000 times */
18     for ( roll = 1; roll <= 6000; roll++ ) {
19         face = 1 + rand() % 6;
20         ++frequency[ face ]; /* replaces 26-line switch of Fig. 5.8 */
21     } /* end for */
22
23     printf( "%s%17s\n", "Face", "Frequency" );
```

Fig. 6.9 | Dice-rolling program using an array instead of switch. (Part I of 2.)



```
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
30  return 0; /* indicates successful termination */
31 } /* end main */
```

Face	Frequency
1	1029
2	951
3	987
4	1033
5	1010
6	990

Fig. 6.9 | Dice-rolling program using an array instead of switch. (Part 2 of 2.)



6.4 Array Examples (Cont.)

- ▶ 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 a `static` 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".



6.4 Array Examples (Cont.)

- ▶ 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.

6.4 Array Examples (Cont.)

- ▶ 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`.

6.4 Array Examples (Cont.)

- ▶ 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( "%s", 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.



6.4 Array Examples (Cont.)

- ▶ 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 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!
- ▶ For this reason, use the conversion specifier `%19s` so that `scanf` does not write characters into memory beyond the end of the array `s`.



6.4 Array Examples (Cont.)

- ▶ 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` reads characters from the keyboard until the first white-space character is encountered—it does not check how large the array is.
- ▶ Thus, `scanf` can write beyond the end of the array.



Common Programming Error 6.7

Not providing `scanf` with a character array large enough to store a string typed at the keyboard can result in destruction of data in a program and other runtime errors. This can also make a system susceptible to worm and virus attacks.



6.4 Array Examples (Cont.)

- ▶ 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.



6.4 Array Examples (Cont.)

- ▶ 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.



```
1  /* Fig. 6.10: fig06_10.c
2     Treating character arrays as strings */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main( void )
7  {
8     char string1[ 20 ]; /* reserves 20 characters */
9     char string2[] = "string literal"; /* reserves 15 characters */
10    int i; /* counter */
11
12    /* read string from user into array string1 */
13    printf("Enter a string: ");
14    scanf( "%s", string1 ); /* input ended by whitespace character */
15
16    /* output strings */
17    printf( "string1 is: %s\nstring2 is: %s\n"
18           "string1 with spaces between characters is:\n",
19           string1, string2 );
20
21    /* output characters until null character is reached */
22    for ( i = 0; string1[ i ] != '\0'; i++ ) {
23        printf( "%c ", string1[ i ] );
24    } /* end for */
```

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



```
25
26     printf( "\n" );
27     return 0; /* indicates successful termination */
28 } /* end main */
```

Enter a string: **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.)



6.4 Array Examples (Cont.)

- ▶ Figure 6.10 uses a **for** statement (line 22) 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, **string1[i] != '\0'**, is true while the terminating null character has not been encountered in the string.



6.4 Array Examples (Cont.)

- ▶ 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.



6.4 Array Examples (Cont.)

- ▶ Arrays that are `static` are initialized once at compile time.
- ▶ If you do not explicitly initialize a `static` array, that array's elements are initialized to zero by the compiler.
- ▶ Figure 6.11 demonstrates function `staticArrayInit` (line 22) with a local `static` array (line 25) and function `automaticArrayInit` (line 44) with a local automatic array (line 47).
- ▶ Function `staticArrayInit` is called twice (lines 12 and 16).
- ▶ The local `static` array in the function is initialized to zero by the compiler (line 25).
- ▶ The function prints the array, adds 5 to each element and prints the array again.



6.4 Array Examples (Cont.)

- ▶ 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 47).
- ▶ 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 6.8

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 */
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     printf( "First call to each function:\n" );
12     staticArrayInit();
13     automaticArrayInit();
14
15     printf( "\n\nSecond call to each function:\n" );
16     staticArrayInit();
17     automaticArrayInit();
18     return 0; /* indicates successful termination */
19 } /* end main */
20
```

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



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

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



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

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 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 4 of 4.)



6.5 Passing Arrays to Functions

- ▶ To pass an array argument to a function, specify the name of the array without any brackets.
- ▶ For example, if array `hourlyTemperatures` has been defined as
 - `int hourlyTemperatures[ 24 ];`the function call
 - `modifyArray( hourlyTemperatures, 24 )`passes array `hourlyTemperatures` and its size to function `modifyArray`.



6.5 Passing Arrays to Functions (Cont.)

- ▶ Unlike `char` arrays that contain strings, other array types do not have a special terminator.
- ▶ For this reason, the size of an array is passed to the function, so that the function can process the proper number of elements.
- ▶ C automatically passes arrays to functions by reference—the called functions can modify the element values in the callers' original arrays.
- ▶ 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.



6.5 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 an 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.



6.5 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 both `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.



Software Engineering Observation 6.2

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



6.5 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 `ints`, `floats` and `chars`) 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.



```
1  /* Fig. 6.12: fig06_12.c
2     The name of an array is the same as &array[ 0 ] */
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     return 0; /* indicates successful termination */
13 } /* 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.



6.5 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.



6.5 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).
- ▶ 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).



6.5 Passing Arrays to Functions (Cont.)

- ▶ Next, `a` and its size are passed to function `modifyArray` (line 27), where each of `a`'s elements is multiplied by 2 (lines 54–55).
- ▶ 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 64) 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.



```
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[], int 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 a */
14     int i; /* counter */
15
16     printf( "Effects of passing entire array by reference:\n\nThe "
17           "values of the original array are:\n" );
18
19     /* output original array */
20     for ( i = 0; i < SIZE; i++ ) {
21         printf( "%3d", a[ i ] );
22     } /* end for */
23 }
```

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



```
24     printf( "\\n" );
25
26     /* pass array a to modifyArray by reference */
27     modifyArray( a, SIZE );
28
29     printf( "The values of the modified array are:\\n" );
30
31     /* output modified array */
32     for ( i = 0; i < SIZE; i++ ) {
33         printf( "%3d", a[ i ] );
34     } /* end for */
35
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     return 0; /* indicates successful termination */
45 } /* end main */
46
```

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



```
47  /* in function modifyArray, "b" points to the original array "a"
48     in memory */
49  void modifyArray( int b[], int size )
50  {
51      int j; /* counter */
52
53      /* multiply each array element by 2 */
54      for ( j = 0; j < size; j++ ) {
55          b[ j ] *= 2;
56      } /* end for */
57  } /* end function modifyArray */
58
59  /* in function modifyElement, "e" is a local copy of array element
60     a[ 3 ] passed from main */
61  void modifyElement( int e )
62  {
63      /* multiply parameter by 2 */
64      printf( "Value in modifyElement is %d\n", e *= 2 );
65  } /* end function modifyElement */
```

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.)



6.5 Passing Arrays to Functions (Cont.)

- ▶ There may be situations in your programs in which a function should not be allowed to modify array elements.
- ▶ Because arrays are always passed by reference, modification of values in an array is difficult to control.
- ▶ C provides the type qualifier `const` 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.



6.5 Passing Arrays to Functions (Cont.)

- ▶ Figure 6.14 demonstrates the `const` qualifier.
- ▶ Function `tryToModifyArray` (line 20) 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 on your system.
- ▶ Each of the three attempts by the function to modify array elements results in the compiler error “`l-value specifies a const object.`”.



```
1  /* Fig. 6.14: fig06_14.c
2     Demonstrating the const type qualifier with arrays */
3  #include <stdio.h>
4
5  void tryToModifyArray( const int b[] ); /* function prototype */
6
7  /* function main begins program execution */
8  int main( void )
9  {
10     int a[] = { 10, 20, 30 }; /* initialize a */
11
12     tryToModifyArray( a );
13
14     printf("%d %d %d\n", a[ 0 ], a[ 1 ], a[ 2 ] );
15     return 0; /* indicates successful termination */
16 } /* end main */
17
18 /* in function tryToModifyArray, array b is const, so it cannot be
19    used to modify the original array a in main. */
20 void tryToModifyArray( const int b[] )
21 {
22     b[ 0 ] /= 2; /* error */
23     b[ 1 ] /= 2; /* error */
24     b[ 2 ] /= 2; /* error */
25 } /* end function tryToModifyArray */
```

Fig. 6.14 | const type qualifier. (Part 1 of 2.)



```
Compiling...
FIG06_14.C
fig06_14.c(22) : error C2166: l-value specifies const object
fig06_14.c(23) : error C2166: l-value specifies const object
fig06_14.c(24) : error C2166: l-value specifies const object
```

Fig. 6.14 | const type qualifier. (Part 2 of 2.)



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. Functions should not be given the capability to modify an array 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.
- ▶ In this chapter we discuss what is perhaps the simplest known sorting scheme.
- ▶ In Chapter 12 and Appendix F, we investigate more complex schemes that yield superior performance.



Performance Tip 6.4

Often, the simplest algorithms perform poorly. Their virtue is that they are easy to write, test and debug. More complex algorithms are often needed to realize maximum performance.



6.6 Sorting Arrays (Cont.)

- ▶ 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 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     This program sorts 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    int i; /* comparisons counter */
13    int hold; /* temporary location used to swap array elements */
14
15    printf( "Data items in original order\n" );
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++ ) {
```

Fig. 6.15 | Sorting an array with bubble sort. (Part I of 3.)



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

Fig. 6.15 | Sorting an array with bubble sort. (Part 2 of 3.)



```
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 with bubble sort. (Part 3 of 3.)



6.6 Sorting Arrays (Cont.)

- ▶ 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, $a[9]$.



6.6 Sorting Arrays (Cont.)

- ▶ 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` loop (lines 24–37).



6.6 Sorting Arrays (Cont.)

- ▶ 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 ];`



6.6 Sorting Arrays (Cont.)

- ▶ 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, the bubble sort 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.



6.7 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.
- ▶ The mean is the arithmetic average of the 99 values.
- ▶ Function **mean** (line 40) computes the mean by totaling the 99 elements and dividing the result by 99.
- ▶ The median is the “middle value.”



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

- ▶ Function `median` (line 61) determines the median by calling function `bubbleSort` (defined in line 133) to sort the array of responses into ascending order, then picking the middle element, `answer[SIZE / 2]`, of the sorted array.
- ▶ When there is an even number of elements, the median should be calculated as the mean of the two middle elements.
- ▶ Function `median` does not currently provide this capability.
- ▶ Function `printArray` (line 156) is called to output the `response` array.
- ▶ The mode is the value that occurs most frequently among the 99 responses.



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

- ▶ Function **mode** (line 82) 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.
- ▶ Figure 6.17 contains a sample run of this program.



```
1  /* Fig. 6.16: fig06_16.c
2     This program introduces the topic of survey data analysis.
3     It computes the mean, median and mode of the data */
4  #include <stdio.h>
5  #define SIZE 99
6
7  /* function prototypes */
8  void mean( const int answer[] );
9  void median( int answer[] );
10 void mode( int freq[], const int answer[] );
11 void bubbleSort( int a[] );
12 void printArray( const int a[] );
13
14 /* function main begins program execution */
15 int main( void )
16 {
17     int frequency[ 10 ] = { 0 }; /* initialize array frequency */
18
19     /* initialize array response */
20     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,
```

Fig. 6.16 | Survey data analysis program. (Part I of 8.)



```
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     return 0; /* indicates successful termination */
37 } /* end main */
38
39 /* calculate average of all response values */
40 void mean( const int answer[] )
41 {
42     int j; /* counter for totaling array elements */
43     int total = 0; /* variable to hold sum of array elements */
44
45     printf( "%s\n%s\n%s\n", "*****", "  Mean", "*****" );
46
```

Fig. 6.16 | Survey data analysis program. (Part 2 of 8.)



```
47  /* total response values */
48  for ( j = 0; j < SIZE; j++ ) {
49      total += answer[ j ];
50  } /* end for */
51
52  printf( "The mean is the average value of the data\n"
53          "items. The mean is equal to the total of\n"
54          "all the data items divided by the number\n"
55          "of data items ( %d ). The mean value for\n"
56          "this run is: %d / %d = %.4f\n\n",
57          SIZE, total, SIZE, ( double ) total / SIZE );
58  } /* end function mean */
59
60  /* sort array and determine median element's value */
61  void median( int answer[] )
62  {
63      printf( "\n%s\n%s\n%s\n%s",
64              "*****", " Median", "*****",
65              "The unsorted array of responses is" );
66
67      printArray( answer ); /* output unsorted array */
68
69      bubbleSort( answer ); /* sort array */
```

Fig. 6.16 | Survey data analysis program. (Part 3 of 8.)



```
70
71     printf( "\n\nThe sorted array is" );
72     printArray( answer ); /* output sorted array */
73
74     /* display median element */
75     printf( "\n\nThe median is element %d of\n"
76             "the sorted %d element array.\n"
77             "For this run the median is %d\n\n",
78             SIZE / 2, SIZE, answer[ SIZE / 2 ] );
79 } /* end function median */
80
81 /* determine most frequent response */
82 void mode( int freq[], const int answer[] )
83 {
84     int rating; /* counter for accessing elements 1-9 of array freq */
85     int j; /* counter for summarizing elements 0-98 of array answer */
86     int h; /* counter for displaying histograms of elements in array freq */
87     int largest = 0; /* represents largest frequency */
88     int modeValue = 0; /* represents most frequent response */
89
90     printf( "\n%s\n%s\n%s\n",
91             "*****", "   Mode", "*****" );
92
```

Fig. 6.16 | Survey data analysis program. (Part 4 of 8.)



```
93  /* initialize frequencies to 0 */
94  for ( rating = 1; rating <= 9; rating++ ) {
95      freq[ rating ] = 0;
96  } /* end for */
97
98  /* summarize frequencies */
99  for ( j = 0; j < SIZE; j++ ) {
100      ++freq[ answer[ j ] ];
101  } /* end for */
102
103  /* output headers for result columns */
104  printf( "%s%11s%19s\n\n%54s\n%54s\n\n",
105          "Response", "Frequency", "Histogram",
106          "1      1      2      2", "5      0      5      0      5" );
107
108  /* output results */
109  for ( rating = 1; rating <= 9; rating++ ) {
110      printf( "%8d%11d", rating, freq[ rating ] );
111
112      /* keep track of mode value and largest frequency value */
113      if ( freq[ rating ] > largest ) {
114          largest = freq[ rating ];
115          modeValue = rating;
116      } /* end if */
```

Fig. 6.16 | Survey data analysis program. (Part 5 of 8.)



```
I117
I118     /* output histogram bar representing frequency value */
I119     for ( h = 1; h <= freq[ rating ]; h++ ) {
I120         printf( "*" );
I121     } /* end inner for */
I122
I123     printf( "\n" ); /* being new line of output */
I124 } /* end outer for */
I125
I126 /* display the mode value */
I127 printf( "The mode is the most frequent value.\n"
I128         "For this run the mode is %d which occurred"
I129         " %d times.\n", modeValue, largest );
I130 } /* end function mode */
I131
```

Fig. 6.16 | Survey data analysis program. (Part 6 of 8.)



```
I32  /* function that sorts an array with bubble sort algorithm */
I33 void bubbleSort( int a[] )
I34 {
I35     int pass; /* pass counter */
I36     int j; /* comparison counter */
I37     int hold; /* temporary location used to swap elements */
I38
I39     /* loop to control number of passes */
I40     for ( pass = 1; pass < SIZE; pass++ ) {
I41
I42         /* loop to control number of comparisons per pass */
I43         for ( j = 0; j < SIZE - 1; j++ ) {
I44
I45             /* swap elements if out of order */
I46             if ( a[ j ] > a[ j + 1 ] ) {
I47                 hold = a[ j ];
I48                 a[ j ] = a[ j + 1 ];
I49                 a[ j + 1 ] = hold;
I50             } /* end if */
I51         } /* end inner for */
I52     } /* end outer for */
I53 } /* end function bubbleSort */
I54
```

Fig. 6.16 | Survey data analysis program. (Part 7 of 8.)



```
155  /* output array contents (20 values per row) */
156  void printArray( const int a[] )
157  {
158      int j; /* counter */
159
160      /* output array contents */
161      for ( j = 0; j < SIZE; j++ ) {
162
163          if ( j % 20 == 0 ) { /* begin new line every 20 values */
164              printf( "\n" );
165          } /* end if */
166
167          printf( "%2d", a[ j ] );
168      } /* end for */
169  } /* end function printArray */
```

Fig. 6.16 | Survey data analysis program. (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: $681 / 99 = 6.8788$

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 8 8 8 8 8 8 8 8
```

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



```
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.)



6.8 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.



6.8 Searching Arrays (Cont.)

- ▶ The linear search (Fig. 6.18) compares each element of the array with the **search key**.
- ▶ Since 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  int linearSearch( const int array[], int key, int size );
8
9  /* function main begins program execution */
10 int main( void )
11 {
12     int a[ SIZE ]; /* create array a */
13     int x; /* counter for initializing elements 0-99 of array a */
14     int searchKey; /* value to locate in array a */
15     int element; /* variable to hold location of searchKey or -1 */
16
17     /* create data */
18     for ( x = 0; x < SIZE; x++ ) {
19         a[ x ] = 2 * x;
20     } /* end for */
21
22     printf( "Enter integer search key:\n" );
23     scanf( "%d", &searchKey );
```

Fig. 6.18 | Linear search of an array. (Part I of 4.)



```
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      printf( "Value not found\n" );
34  } /* end else */
35
36  return 0; /* indicates successful termination */
37 } /* end main */
38
```

Fig. 6.18 | Linear search of an array. (Part 2 of 4.)



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

Fig. 6.18 | Linear search of an array. (Part 3 of 4.)



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 4 of 4.)



6.8 Searching Arrays (Cont.)

- ▶ 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.



6.8 Searching Arrays (Cont.)

- ▶ If they are equal, the search key is found and the array subscript of that element is returned.
- ▶ If they are 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.



6.8 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 1024 by 2 yields the values 512, 256, 128, 64, 32, 16, 8, 4, 2 and 1.
- ▶ The number 1024 (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.



6.8 Searching Arrays (Cont.)

- ▶ An array of 1048576 (2^{20}) elements takes a maximum of 20 comparisons to find the search key.
- ▶ An array of one billion elements takes a maximum of 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!



6.8 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` (lines 44–74).
- ▶ 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.



6.8 Searching Arrays (Cont.)

- ▶ 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 a maximum of 4 comparisons are required to find the search key.



6.8 Searching Arrays (Cont.)

- ▶ The program uses function `printHeader` (lines 77–96) to output the array subscripts and function `printRow` (lines 100–120) 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  int binarySearch( const int b[], int searchKey, int low, int high );
8  void printHeader( void );
9  void printRow( const int b[], int low, int mid, int high );
10
11 /* function main begins program execution */
12 int main( void )
13 {
14     int a[ SIZE ]; /* create array a */
15     int i; /* counter for initializing elements 0-14 of array a */
16     int key; /* value to locate in array a */
17     int 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 }
```

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



```
24     printf( "Enter a number between 0 and 28: " );
25     scanf( "%d", &key );
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
40     return 0; /* indicates successful termination */
41 } /* end main */
42
```

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



```
43  /* function to perform binary search of an array */
44  int binarySearch( const int b[], int searchKey, int low, int high )
45  {
46      int middle; /* variable to hold middle element of array */
47
48      /* loop until low subscript is greater than high subscript */
49      while ( low <= high ) {
50
51          /* determine middle element of subarray being searched */
52          middle = ( low + high ) / 2;
53
54          /* display subarray used in this loop iteration */
55          printRow( b, low, middle, high );
56
57          /* if searchKey matched middle element, return middle */
58          if ( searchKey == b[ middle ] ) {
59              return middle;
60          } /* end if */
61
62          /* if searchKey less than middle element, set new high */
63          else if ( searchKey < b[ middle ] ) {
64              high = middle - 1; /* search low end of array */
65          } /* end else if */
66      }
```

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



```
67     /* if searchKey greater than middle element, set new low */
68     else {
69         low = middle + 1; /* search high end of array */
70     } /* end else */
71 } /* end while */
72
73     return -1; /* searchKey not found */
74 } /* end function binarySearch */
75
76 /* Print a header for the output */
77 void printHeader( void )
78 {
79     int i; /* counter */
80
81     printf( "\nSubscripts:\n" );
82
83     /* output column head */
84     for ( i = 0; i < SIZE; i++ ) {
85         printf( "%3d ", i );
86     } /* end for */
87
88     printf( "\n" ); /* start new line of output */
89
```

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



```
90     /* output line of - characters */
91     for ( i = 1; i <= 4 * SIZE; i++ ) {
92         printf( "-" );
93     } /* end for */
94
95     printf( "\n" ); /* start new line of output */
96 } /* end function printHeader */
97
98 /* Print one row of output showing the current
99    part of the array being processed. */
100 void printRow( const int b[], int low, int mid, int high )
101 {
102     int i; /* counter for iterating through array b */
103
104     /* loop through entire array */
105     for ( i = 0; i < SIZE; i++ ) {
106
107         /* display spaces if outside current subarray range */
108         if ( i < low || i > high ) {
109             printf( "    " );
110         } /* end if */
111         else if ( i == mid ) { /* display middle element */
112             printf( "%3d*", b[ i ] ); /* mark middle value */
113         } /* end else if */
114     }
```

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



```
114     else { /* display other elements in subarray */
115         printf( "%3d ", b[ i ] );
116     } /* end else */
117 } /* end for */
118
119 printf( "\n" ); /* start new line of output */
120 } /* 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 6 of 7.)



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 7 of 7.)



6.9 Multiple-Subscripted Arrays

- ▶ Arrays in C can have multiple subscripts.
- ▶ A common use of **multiple-subscripted arrays** (also called **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**.



6.9 Multiple-Subscripted Arrays (Cont.)

- ▶ Multiple-subscripted 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***

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Diagram illustrating the structure of a double-subscripted array with three rows and four columns. The array is represented as a 3x4 grid of elements. The rows are labeled Row 0, Row 1, and Row 2. The columns are labeled Column 0, Column 1, Column 2, and Column 3. Each element is shown as a light blue box containing its address, e.g., a[0][0].

Arrows indicate the indexing structure for the element a[2][1]:

- Column index: points to the '1' in the second subscript.
- Row index: points to the '2' in the first subscript.
- Array name: points to the 'a' in the first subscript.

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



6.9 Multiple-Subscripted 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 subscripts that uniquely identify each element in **a**.
- ▶ The names of the elements in the first row all have a first subscript of 0; the names of the elements in the fourth column all have a second subscript of 3.



Common Programming Error 6.9

Referencing a double-subscripted array element as $a[x, y]$ instead of $a[x][y]$. C interprets $a[x, y]$ as $a[y]$, and as such it does not cause a compilation error.

6.9 Multiple-Subscripted Arrays (Cont.)

- ▶ A multiple-subscripted 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.



6.9 Multiple-Subscripted 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.



6.9 Multiple-Subscripted Arrays (Cont.)

- ▶ Figure 6.21 demonstrates defining and initializing double-subscripted arrays.
- ▶ 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 the first row (i.e., row 0) of the array to the values 1, 2 and 3; and the second sublist initializes the second row (i.e., row 1) of the array to the values 4, 5 and 6.



```
1  /* Fig. 6.21: fig06_21.c
2     Initializing multidimensional arrays */
3  #include <stdio.h>
4
5  void printArray( const 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     printf( "Values in array1 by row are:\n" );
16     printArray( array1 );
17
18     printf( "Values in array2 by row are:\n" );
19     printArray( array2 );
20
21     printf( "Values in array3 by row are:\n" );
22     printArray( array3 );
23     return 0; /* indicates successful termination */
24 } /* end main */
```

Fig. 6.21 | Initializing multidimensional arrays. (Part I of 3.)



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

Fig. 6.21 | Initializing multidimensional arrays. (Part 2 of 3.)



```
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 3 of 3.)



6.9 Multiple-Subscripted 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` (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.



6.9 Multiple-Subscripted 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` (lines 27–43) 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.



6.9 Multiple-Subscripted Arrays (Cont.)

- ▶ The first subscript of a multiple-subscripted 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 multiple-subscripted 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.



6.9 Multiple-Subscripted Arrays (Cont.)

- ▶ 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 the third element of that row (element 2).



6.9 Multiple-Subscripted Arrays (Cont.)

- ▶ Many common array manipulations use **for** repetition statements.
- ▶ For example, the following statement sets all the elements in the third row of array **a** in Fig. 6.20 to zero:
 - **for** (**column** = **0**; **column** <= **3**; **column**++) {
 a[**2**][**column**] = **0**;
}
- ▶ We specified the third row, therefore we know that the first subscript is always 2 (again, 0 is the first row and 1 is the second).



6.9 Multiple-Subscripted Arrays (Cont.)

- ▶ The `loop` varies only the second subscript (i.e., the column).
- ▶ 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;
```

## 6.9 Multiple-Subscripted Arrays (Cont.)

- ▶ 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.



## 6.9 Multiple-Subscripted Arrays (Cont.)

- ▶ The outer **for** statement begins by setting **row** (i.e., the row subscript) to **0** so that the elements of the first row may be totaled by the inner **for** statement.
- ▶ The outer **for** statement then increments **row** to **1**, so the elements of the second row can be totaled.
- ▶ Then, the outer **for** statement increments **row** to **2**, so the elements of the third row can be totaled.
- ▶ The result is printed when the nested **for** statement terminates.



## 6.9 Multiple-Subscripted Arrays (Cont.)

- ▶ 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 43–62) determines the lowest grade of any student for the semester.



## 6.9 Multiple-Subscripted Arrays (Cont.)

- ▶ Function `maximum` (lines 65–84) determines the highest grade of any student for the semester.
- ▶ Function `average` (lines 87–98) determines a particular student's semester average.
- ▶ Function `printArray` (lines 101–120) outputs the double-subscripted array in a neat, tabular format.



```
1 /* Fig. 6.22: fig06_22.c
2 Double-subscripted array example */
3 #include <stdio.h>
4 #define STUDENTS 3
5 #define EXAMS 4
6
7 /* function prototypes */
8 int minimum(const int grades[][EXAMS], int pupils, int tests);
9 int maximum(const int grades[][EXAMS], int pupils, int tests);
10 double average(const int setOfGrades[], int tests);
11 void printArray(const int grades[][EXAMS], int pupils, int tests);
12
13 /* function main begins program execution */
14 int main(void)
15 {
16 int student; /* student counter */
17
18 /* initialize student grades for three students (rows) */
19 const int studentGrades[STUDENTS][EXAMS] =
20 { { 77, 68, 86, 73 },
21 { 96, 87, 89, 78 },
22 { 70, 90, 86, 81 } };
23 }
```

**Fig. 6.22** | Double-subscripted arrays example. (Part I of 7.)



```
24 /* output array studentGrades */
25 printf("The array is:\n");
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 %d is %.2f\n",
36 student, average(studentGrades[student], EXAMS));
37 } /* end for */
38
39 return 0; /* indicates successful termination */
40 } /* end main */
41
```

**Fig. 6.22** | Double-subscripted arrays example. (Part 2 of 7.)



```
42 /* Find the minimum grade */
43 int minimum(const int grades[][EXAMS], int pupils, int tests)
44 {
45 int i; /* student counter */
46 int j; /* exam counter */
47 int lowGrade = 100; /* initialize to highest possible grade */
48
49 /* loop through rows of grades */
50 for (i = 0; i < pupils; i++) {
51
52 /* loop through columns of grades */
53 for (j = 0; j < tests; j++) {
54
55 if (grades[i][j] < lowGrade) {
56 lowGrade = grades[i][j];
57 } /* end if */
58 } /* end inner for */
59 } /* end outer for */
60
61 return lowGrade; /* return minimum grade */
62 } /* end function minimum */
63
```

**Fig. 6.22** | Double-subscripted arrays example. (Part 3 of 7.)



```
64 /* Find the maximum grade */
65 int maximum(const int grades[][EXAMS], int pupils, int tests)
66 {
67 int i; /* student counter */
68 int j; /* exam counter */
69 int highGrade = 0; /* initialize to lowest possible grade */
70
71 /* loop through rows of grades */
72 for (i = 0; i < pupils; i++) {
73
74 /* loop through columns of grades */
75 for (j = 0; j < tests; j++) {
76
77 if (grades[i][j] > highGrade) {
78 highGrade = grades[i][j];
79 } /* end if */
80 } /* end inner for */
81 } /* end outer for */
82
83 return highGrade; /* return maximum grade */
84 } /* end function maximum */
85
```

**Fig. 6.22** | Double-subscripted arrays example. (Part 4 of 7.)



```
86 /* Determine the average grade for a particular student */
87 double average(const int setOfGrades[], int tests)
88 {
89 int i; /* exam counter */
90 int total = 0; /* sum of test grades */
91
92 /* total all grades for one student */
93 for (i = 0; i < tests; i++) {
94 total += setOfGrades[i];
95 } /* end for */
96
97 return (double) total / tests; /* average */
98 } /* end function average */
99
```

**Fig. 6.22** | Double-subscripted arrays example. (Part 5 of 7.)



```
100 /* Print the array */
101 void printArray(const int grades[][EXAMS], int pupils, int tests)
102 {
103 int i; /* student counter */
104 int j; /* exam counter */
105
106 /* output column heads */
107 printf(" [0] [1] [2] [3]");
108
109 /* output grades in tabular format */
110 for (i = 0; i < pupils; i++) {
111
112 /* output label for row */
113 printf("\nstudentGrades[%d] ", i);
114
115 /* output grades for one student */
116 for (j = 0; j < tests; j++) {
117 printf("%-5d", grades[i][j]);
118 } /* end inner for */
119 } /* end outer for */
120 } /* end function printArray */
```

**Fig. 6.22** | Double-subscripted arrays example. (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** | Double-subscripted arrays example. (Part 7 of 7.)



## 6.9 Multiple-Subscripted 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.

## 6.9 Multiple-Subscripted Arrays (Cont.)

- ▶ 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 */
```



## 6.9 Multiple-Subscripted Arrays (Cont.)

- ▶ The outer **for** statement begins by setting **i** (i.e., the row subscript) to 0 so that the elements of the first 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 the second row are compared to variable **lowGrade**.



## 6.9 Multiple-Subscripted Arrays (Cont.)

- ▶ The outer **for** statement then increments the row subscript to 2.
- ▶ The elements of the third row are compared to variable **lowGrade**.
- ▶ When execution of the nested structure is complete, **lowGrade** contains the smallest grade in the double-subscripted array.
- ▶ Function **maximum** works similarly to function **minimum**.
- ▶ Function **average** (line 87) takes two arguments—a single-subscripted array of test results for a particular student called **setOfGrades** and the number of test results in the array.



## 6.9 Multiple-Subscripted Arrays (Cont.)

- ▶ When **average** is called, the first argument **studentGrades[student]** is passed.
- ▶ This causes the address of one row of the double-subscripted array to be passed to **average**.
- ▶ The argument **studentGrades[1]** is the starting address of the second row of the array.
- ▶ Remember that a double-subscripted array is basically an array of single-subscripted arrays and that the name of a single-subscripted 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.