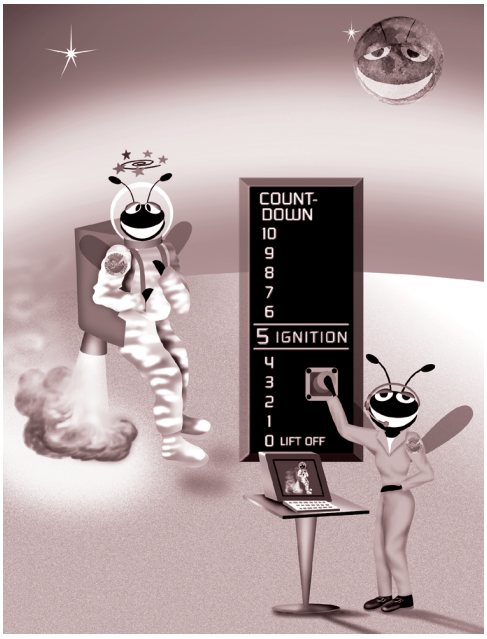
6

C Arrays

Objectives

- To introduce the array data structure.
- To understand the use of arrays to store, sort and search lists and tables of values.
- To understand how to declare an array, initialize an array and refer to individual elements of an array.
- To be able to pass arrays to functions.
- To understand basic sorting techniques.
- To be able to declare and manipulate multiple subscript arrays.



With sobs and tears he sorted out
Those of the largest size …
Lewis Carroll

Attempt the end, and never stand to doubt;
Nothing's so hard, but search will find it out.
Robert Herrick

Now go, write it before them in a table,
and note it in a book.
Isaiah 30:8

'Tis in my memory lock'd,
And you yourself shall keep the key of it.
William Shakespeare

Outline

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 (they may, of course, be of automatic storage class and hence created and destroyed each time the blocks in which they are defined are entered and exited). 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 (**`[]`**). 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 letter, digit and underscore characters. Array names cannot begin with a digit character.

The position number contained within square brackets is more formally called a *subscript.* A subscript must be an integer or an integer expression. 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;
```

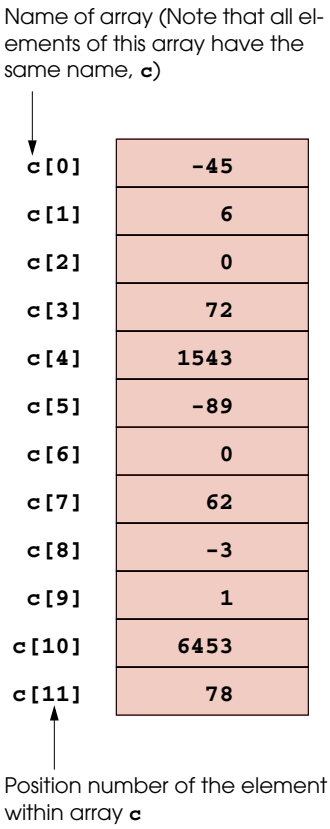


Fig. 6.1 A 12-element array.

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

Let us examine array `c` in Fig. 6.1 more closely. The *name* of the array 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 would 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 would write

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

Common Programming Error 6.1

It is 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.

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 parentheses. Figure 6.2 shows the precedence and associativity of the operators introduced to this point in the text. They are shown top to bottom in decreasing order of precedence.

6.3 Declaring Arrays

Arrays occupy space in memory. The programmer specifies 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**, the declaration

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

is used. Memory may be reserved for several arrays with a single declaration. To reserve 100 elements for integer array **b** and 27 elements for integer array **x**, the following declaration is used:

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

Arrays may be declared to contain other data types. For example, an array of type **char** can be used to store a character string. Character strings and their similarity to arrays are discussed in Chapter 8. The relationship between pointers and arrays is discussed in Chapter 7.

6.4 Examples Using Arrays

Figure 6.3 uses a **for** repetition structure to initialize the elements of a 10-element integer array **n** to zeros and prints the array in tabular format.

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.

```
1  /* Fig. 6.3: fig06_03.c
2     initializing an array */
3  #include <stdio.h>
4
5  int main()
6  {
7     int n[ 10 ], i;
8
9     for ( i = 0; i <= 9; i++ )       /* initialize array */
10       n[ i ] = 0;
11
12    printf( "%s%13s\n", "Element", "Value" );
13    for ( i = 0; i <= 9; i++ )       /* print array */
14       printf( "%7d%13d\n", i, n[ i ] );
15
16    return 0;
17 }
```

```
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.

Note that we chose not to place a blank line between the first **printf** statement (line 12) and the **for** structure in Fig. 6.3 because they are closely related. In this case, the **printf** statement displays the column heads for the two columns printed in the **for** structure. Programmers often omit the blank line between a **for** structure and a closely related **printf** statement.

The elements of an array can also be initialized in the array declaration by following the declaration with an equal sign and a comma-separated list (enclosed in braces) of *initializers*. Figure 6.4 initializes an integer array with ten values (line 7) and prints the array in tabular format.

```
1  /* Fig. 6.4: fig06_04.c
2     Initializing an array with a declaration */
3  #include <stdio.h>
4
5  int main()
6  {
7     int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
```

Fig. 6.4 Initializing the elements of an array with a declaration (part 1 of 2).

```
 8     int i;
 9
10     printf( "%s%13s\n", "Element", "Value" );
11
12     for ( i = 0; i <= 9; i++ )
13        printf( "%7d%13d\n", i, n[ i ] );
14
15     return 0;
16  }
```

```
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 a declaration (part 2 of 2).

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 with the declaration

```
int n[ 10 ] = { 0 };
```

which explicitly initializes the first element to zero and initializes the remaining nine elements to zero because there are fewer initializers than there are elements in the array. It is important to remember that arrays are not automatically initialized to zero. The programmer 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 run time for automatic arrays.

Common Programming Error 6.2

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

The following array declaration

```
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.

If the array size is omitted from a declaration 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.

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**.

The **#define** preprocessor directive is introduced in this program. Line 5

```
#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. 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, the first **for** loop (line 11) could 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 would have to change the program in three separate places to scale the program to handle 1000 array elements. As programs get larger, this technique becomes more useful for writing clear programs..

```
1  /* Fig. 6.5: fig06_05.c
2     Initialize the elements of array s to
3     the even integers from 2 to 20 */
4  #include <stdio.h>
5  #define SIZE 10
6
7  int main()
8  {
9     int s[ SIZE ], j;
10
11    for ( j = 0; j <= SIZE - 1; j++ )   /* set the values */
12       s[ j ] = 2 + 2 * j;
13
14    printf( "%s%13s\n", "Element", "Value" );
15
16    for ( j = 0; j < SIZE; j++ )   /* print the values */
17       printf( "%7d%13d\n", j, s[ j ] );
18
19    return 0;
20 }
```

```
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 Generating the values to be placed into elements of an array.

Common Programming Error 6.4

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

If the preceding **`#define`** preprocessor directive 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. Remember that the preprocessor is not C—it is only a text manipulator.

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 the programmer that symbolic constants are not variables.

Figure 6.6 sums the values contained in the 12-element integer array **`a`**. The **`for`** loop's body (line 13) does the totaling.

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.*

```
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  int main()
7  {
8     int a[ SIZE ] = { 1, 3, 5, 4, 7, 2, 99,
9                       16, 45, 67, 89, 45 };
10    int i, total = 0;
11
12    for ( i = 0; i <= SIZE - 1; i++ )
13       total += a[ i ];
14
15    printf( "Total of array element values is %d\n", total );
16    return 0;
17 }
```

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

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

This is a typical array application (see Fig. 6.7). We wish to summarize the number of responses of each type (i.e., 1 through 10). The array **responses** (line 10) is a 40-element array of the students' responses. We use an 11-element array, **frequency** (line 9) to count the number of occurrences of each response. We ignore **frequency[0]**, because it is logical to have the response 1 increment **frequency[1]** than **frequency[0]**. This allows us to use each response directly as the subscript in the **frequency** array.

Good Programming Practice 6.2

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.

```
1  /* Fig. 6.7: fig06_07.c
2     Student poll program */
3  #include <stdio.h>
4  #define RESPONSE_SIZE 40
5  #define FREQUENCY_SIZE 11
6
7  int main()
8  {
9     int answer, rating, frequency[ FREQUENCY_SIZE ] = { 0 };
10    int responses[ RESPONSE_SIZE ] =
11       { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
12         1, 6, 3, 8, 6, 10, 3, 8, 2, 7,
13         6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
14         5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
15
16    for ( answer = 0; answer <= RESPONSE_SIZE - 1; answer++ )
17       ++frequency[ responses [ answer ] ];
18
19    printf( "%s%17s\n", "Rating", "Frequency" );
20
21    for ( rating = 1; rating <= FREQUENCY_SIZE - 1; rating++ )
22       printf( "%6d%17d\n", rating, frequency[ rating ] );
23
24    return 0;
25 }
```

```
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 A simple student poll analysis program.

Performance Tip 6.1

Sometimes performance considerations far outweigh clarity considerations.

The **for** loop (line 16) takes the responses one at a time from the array **response** and increments one of the 10 counters (**frequency[1]** to **frequency[10]**) in the **frequency** array. The key statement in the loop is line 17

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

This statement increments the appropriate **frequency** counter depending on the value of **responses[answer]**. For example, when the counter variable **answer** is **0**, **responses[answer]** is **1**, so **++frequency[responses[answer]];** is actually interpreted as

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

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

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

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

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

which increments array element six, and so on. Note that regardless of the number of responses processed in the survey, only an 11-element array is required (ignoring element zero) to summarize the results. If the data contained invalid values such as 13, the program would attempt to add **1** to **frequency[13]**. This would be outside the bounds of the array. *C has no array bounds checking to prevent the computer from referring to an element that does not exist.* Thus, an executing program can walk off the end of an array without warning. The programmer 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.

Good Programming Practice 6.3

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

Good Programming Practice 6.4

Mention the high array subscript in a **for** *structure to help eliminate off-by-one errors.*

Good Programming Practice 6.5

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

Performance Tip 6.2

The (normally serious) effects of referencing elements outside the array bounds are system dependent.

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`** loop actually draws the bars. Note the use of **`printf( "\n" )`** to end a histogram bar.

In Chapter 5, we stated that we would show a more elegant method of writing the dice-rolling program of Fig. 5.8. The problem was to 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. 6.8: fig06_08.c
2     Histogram printing program */
3  #include <stdio.h>
4  #define SIZE 10
5
6  int main()
7  {
8     int n[ SIZE ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
9     int i, j;
10
11    printf( "%s%13s%17s\n", "Element", "Value", "Histogram" );
12
13    for ( i = 0; i <= SIZE - 1; i++ ) {
14       printf( "%7d%13d        ", i, n[ i ]) ;
15
16       for ( j = 1; j <= n[ i ]; j++ )   /* print one bar */
17          printf( "%c", '*' );
18
19       printf( "\n" );
20    }
21
22    return 0;
23 }
```

```
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 A program that prints histograms.

```
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  int main()
9  {
10    int face, roll, frequency[ SIZE ] = { 0 };
11
12    srand( time( NULL ) );
13
14    for ( roll = 1; roll <= 6000; roll++ ) {
15       face = rand() % 6 + 1;
16       ++frequency[ face ];     /* replaces 20-line switch */
17    }                           /* of Fig. 5.8 */
18
19    printf( "%s%17s\n", "Face", "Frequency" );
20
21    for ( face = 1; face <= SIZE - 1; face++ )
22       printf( "%4d%17d\n", face, frequency[ face ] );
23
24    return 0;
25 }
```

```
Face        Frequency
   1             1037
   2              987
   3             1013
   4             1028
   5              952
   6              983
```

Fig. 6.9 Dice-rolling program using arrays instead of **`switch`**.

To this point we have discussed only integer arrays. However, arrays are capable of holding data of any type. We now discuss storing strings in character arrays. So far, the only string processing capability we have is outputting a string with **`printf`**. A string such as "**`hello`**" is really a **`static`** array of individual characters in C.

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

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

initializes the elements of array **`string1`** to the individual characters in the string literal **`"first"`**. The size of array **`string1`** in the preceding declaration is determined by the compiler based on the length of the string. It is important to note that 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 repre-

sentation of the null character is **'\0'**. All strings in C end with this character. A character array representing a string should always be declared 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. The preceding declaration 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 specification **%s**. For example, the declaration

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

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

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

reads a string from the keyboard into **string2**. Note that the name of the array is passed to **scanf** without the preceding **&** used with other variables. The **&** is normally used to provide **scanf** with a variable's location in memory so a value can be stored there. In Section 6.5, we discuss passing arrays to functions. We will see that an array name is the address of the start of the array; therefore, the **&** is not necessary.

It is the programmer's 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 whitespace character is encountered—it does not care 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 loss of data in a program and other run-time errors.*

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 );
```

Note that **printf**, like **scanf**, does not care how large the character array is. The characters of the string are printed until a terminating null character is encountered.

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.

Figure 6.10 uses a **for** structure (line 16) to loop through the **string1** array and print the individual characters separated by spaces using the **%c** conversion specification. The condition in the **for** structure, **string1[i] != '\0'**, is true while the terminating null character has not been encountered in the string.

```
1  /* Fig. 6.10: fig06_10.c
2     Treating character arrays as strings */
3  #include <stdio.h>
4
5  int main()
6  {
7     char string1[ 20 ], string2[] = "string literal";
8     int i;
9
10    printf(" Enter a string: ");
11    scanf( "%s", string1 );
12    printf( "string1 is: %s\nstring2: is %s\n"
13            "string1 with spaces between characters is:\n",
14            string1, string2 );
15
16    for ( i = 0; string1[ i ] != '\0'; i++ )
17       printf( "%c ", string1[ i ] );
18
19    printf( "\n" );
20    return 0;
21 }
```

```
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.

Chapter 5 discussed the storage class specifier **static**. A **static** local variable exists for the duration of the program, but is only visible in the function body. We can apply **static** to a local array declaration 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 6.3

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

Arrays that are declared **static** are automatically initialized once at compile time. If a **static** array is not explicitly initialized by the programmer, that array's elements are initialized to zero by the compiler.

Figure 6.11 demonstrates function **staticArrayInit** (line 20) with a local array declared **static** and function **automaticArrayInit** (line 37) with an automatic local array. Function **staticArrayInit** is called twice (lines 11 and 14). The **static** local array in the function is initialized to zero by the compiler. The function prints the array, adds 5 to each element and prints the array again. The second time the function is called, the **static** array contains the values stored during the first function call. Function **automaticArrayInit** is also called twice (lines 12 and 15). The elements of the auto-

matic 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 6.8

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

```
1  /* Fig. 6.11: fig06_11.c
2     Static arrays are initialized to zero */
3  #include <stdio.h>
4
5  void staticArrayInit( void );
6  void automaticArrayInit( void );
7
8  int main()
9  {
10    printf( "First call to each function:\n" );
11    staticArrayInit();
12    automaticArrayInit();
13    printf( "\n\nSecond call to each function:\n" );
14    staticArrayInit();
15    automaticArrayInit();
16    return 0;
17 }
18
19 /* function to demonstrate a static local array */
20 void staticArrayInit( void )
21 {
22    static int a[ 3 ];
23    int i;
24
25    printf( "\nValues on entering staticArrayInit:\n" );
26
27    for ( i = 0; i <= 2; i++ )
28       printf( "array1[%d] = %d  ", i, a[ i ] );
29
30    printf( "\nValues on exiting staticArrayInit:\n" );
31
32    for ( i = 0; i <= 2; i++ )
33       printf( "array1[%d] = %d  ", i, a[ i ] += 5 );
34 }
35
36 /* function to demonstrate an automatic local array */
37 void automaticArrayInit( void )
38 {
39    int a[ 3 ] = { 1, 2, 3 }, i;
40
41    printf( "\n\nValues on entering automaticArrayInit:\n" );
42
```

Fig. 6.11 Static arrays are automatically initialized to zero if not explicitly initialized by the programmer (part 1 of 2).

```
43      for ( i = 0; i <= 2; i++ )
44         printf("array1[ %d ] = %d  ", i, a[ i ] );
45
46      printf( "\nValues on exiting automaticArrayInit:\n" );
47
48      for ( i = 0; i <= 2; i++ )
49         printf( "array1[ %d ] = %d  ", i, a[ i ] += 5 );
50   }
```

```
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:
array1[0] = 1  array1[1] = 2  array1[2] = 3
Values on exiting automaticArrayInit:
array1[0] = 6  array1[1] = 7  array1[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:
array1[0] = 1  array1[1] = 2  array1[2] = 3
Values on exiting automaticArrayInit:
array1[0] = 6  array1[1] = 7  array1[2] = 8
```

Fig. 6.11 Static arrays are automatically initialized to zero if not explicitly initialized by the programmer (part 2 of 2).

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 declared as

```
int hourlyTemperatures[ 24 ];
```

the function call statement

```
modifyArray( hourlyTemperatures, 24 );
```

passes array **hourlyTemperatures** and its size to function **modifyArray**. When passing an array to a function, the array size is often passed so the function can process the specific number of elements in the array.

C automatically passes arrays to functions using simulated call by reference—the called functions can modify the element values in the callers' original arrays. The name of the array is actually the address of the first element of the array! Because the starting

address of the array is passed, the called function knows precisely where the array is stored. Therefore, when the called function modifies array elements in its function body, it is 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 specification—a special conversion specification for printing addresses. The **`%p`** conversion specification normally outputs addresses as hexadecimal numbers. Hexadecimal (base 16) numbers consist of the digits 0 through 9 and the letters A through F. They are often used as shorthand notation for large integer values. Appendix E, "Number Systems," provides an in-depth discussion of the relationships between 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 **`0065FDF0`**. The output of this program is system dependent, but the addresses are always identical.

Performance Tip 6.4

Passing arrays simulated call by reference makes sense for performance reasons. If arrays were passed call by value, a copy of each element would be passed. For large, frequently passed arrays, this would be time consuming and would consume considerable storage for the copies of the arrays.

Software Engineering Observation 6.2

It is possible to pass an array by value by using a simple trick we explain in Chapter 10.

Although entire arrays are passed simulated call by reference, individual array elements are passed call by value exactly as simple variables are. Such simple single pieces of data are called *scalars* or *scalar quantities.* 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 simulate call by reference for scalars (i.e., individual variables and array elements).

```
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  int main()
6  {
7     char array[ 5 ];
8
9     printf( "    array = %p\n&array[0] = %p\n"
10            "   &array = %p\n",
11           array, &array[ 0 ], &array );
12    return 0;
13 }
```

```
    array = 0065FDF0
&array[0] = 0065FDF0
   &array = 0065FDF0
```

Fig. 6.12 The name of an array is the same as the address of the array's first element.

For a function to receive an array through a function call, the function's parameter list must specify that an array will be received. For example, the function header for function **modifyArray** might be written as

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

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

Note the strange appearance of the function prototype for **modifyArray**

```
void modifyArray( int [], int );
```

This prototype could have been written

```
void modifyArray( int anyArrayName[], int anyVariableName );
```

but as we learned in Chapter 5, the C compiler ignores variable names in prototypes.

Good Programming Practice 6.6

Some programmers include variable names in function prototypes to make programs clearer. The compiler ignores these names.

Remember, the prototype tells the compiler the number of arguments and the types of each argument (in the order in which the arguments are executed) to appear.

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 17 and 18). Next, **a** and its size are passed to function **modifyArray** (defined in line 34) where each of **a**'s elements is multiplied by 2. Then **a** is reprinted in **main** in line 28. 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** (defined in line 42). Function **modifyElement** multiplies its argument by 2 and prints the new value. Note that when **a[3]** is reprinted in **main**, it has not been modified because individual array elements are passed call 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  void modifyArray( int [], int );  /* appears strange */
7  void modifyElement( int );
```

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

```
 8
 9  int main()
10  {
11     int a[ SIZE ] = { 0, 1, 2, 3, 4 }, i;
12
13     printf( "Effects of passing entire array call "
14             "by reference:\n\nThe values of the "
15             "original array are:\n" );
16
17     for ( i = 0; i <= SIZE - 1; i++ )
18        printf( "%3d", a[ i ] );
19
20     printf( "\n" );
21     modifyArray( a, SIZE );  /* passed call by reference */
22     printf( "The values of the modified array are:\n" );
23
24     for ( i = 0; i <= SIZE - 1; i++ )
25        printf( "%3d", a[ i ] );
26
27     printf( "\n\n\nEffects of passing array element call "
28             "by value:\n\nThe value of a[3] is %d\n", a[ 3 ] );
29     modifyElement( a[ 3 ] );
30     printf( "The value of a[ 3 ] is %d\n", a[ 3 ] );
31     return 0;
32  }
33
34  void modifyArray( int b[], int size )
35  {
36     int j;
37
38     for ( j = 0; j <= size - 1; j++ )
39        b[ j ] *= 2;
40  }
41
42  void modifyElement( int e )
43  {
44     printf( "Value in modifyElement is %d\n", e *= 2 );
45  }
```

```
Effects of passing entire array call 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 call by value:


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

Fig. 6.13 Passing arrays and individual array elements to functions (part 2 of 2).

There may be situations in your programs in which a function should not be allowed to modify array elements. Because arrays are always passed simulated call 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 elements of the array 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 the programmer to correct a program so it does not attempt to modify array elements. Although the **const** qualifier is well defined in the ANSI standard, C systems vary in their ability to enforce it.

Figure 6.14 demonstrates the **const** qualifier. Function **tryToModifyArray** (line 16) 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 "**Cannot modify a const object**." The **const** qualifier is discussed again in Chapter 7.

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 is absolutely necessary.*

```
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 [] );
6
7  int main()
8  {
9     int a[] = { 10, 20, 30 };
10
11    tryToModifyArray( a );
12    printf("%d %d %d\n", a[ 0 ], a[ 1 ], a[ 2 ] );
13    return 0;
14 }
15
16 void tryToModifyArray( const int b[] )
17 {
18    b[ 0 ] /= 2;     /* error */
19    b[ 1 ] /= 2;     /* error */
20    b[ 2 ] /= 2;     /* error */
21 }
```

```
Compiling...
Fig06_14.c(18) : error C2166: l-value specifies const object
Fig06_14.c(19) : error C2166: l-value specifies const object
Fig06_14.c(20) : error C2166: l-value specifies const object
```

Fig. 6.14 Demonstrating the **const** type qualifier.

6.6 Sorting Arrays

Sorting data (i.e., placing the data into a particular order such as ascending or descending) is one of the most important computing applications. A bank sorts all checks by account number so that it can prepare individual bank statements at the end of each month. Telephone companies sort their lists of accounts by last name and, within that, by first name to make it easy to find phone numbers. Virtually every organization must sort some data and in many cases massive amounts of data. Sorting data is an intriguing problem which has attracted some of the most intense research efforts in the field of computer science. In this chapter we discuss what is perhaps the simplest known sorting scheme. In the exercises and in Chapter 12, we investigate more complex schemes that yield far superior performance.

Performance Tip 6.5

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

Figure 6.15 sorts the values in the elements of the 10-element array **a** (line 9) 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
3     ascending order */
4  #include <stdio.h>
5  #define SIZE 10
6
7  int main()
8  {
9     int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
10    int i, pass, hold;
11
12    printf( "Data items in original order\n" );
13
14    for ( i = 0; i <= SIZE - 1; i++ )
15       printf( "%4d", a[ i ] );
16
17    for ( pass = 1; pass <= SIZE - 1; pass++ )  /* passes */
18
19       for ( i = 0; i <= SIZE - 2; i++ )       /* one pass */
20
21          if ( a[ i ] > a[ i + 1 ] ) {  /* one comparison */
22             hold = a[ i ];                    /* one swap */
23             a[ i ] = a[ i + 1 ];
24             a[ i + 1 ] = hold;
25          }
```

Fig. 6.15 Sorting an array with bubble sort (part 1 of 2).

```
26
27     printf( "\nData items in ascending order\n" );
28
29     for ( i = 0; i <= SIZE - 1; i++ )
30        printf( "%4d", a[ i ] );
31
32     printf( "\n" );
33
34     return 0;
35  }
```

```
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 2 of 2).

First the program compares **a[0]** to **a[1]**, then **a[1]** to **a[2]**, then **a[2]** to **a[3]**, and so on until it completes the pass by comparing **a[8]** to **a[9]**. Note that although there are 10 elements, only nine comparisons are performed. Because of the way the successive comparisons are made, a large value may move down the array many positions on a single pass, but a small value may move up only one position. On the first pass, the largest value is guaranteed to sink to the bottom element of the array, **a[9]**. On the second pass, the second largest value is guaranteed to sink to **a[8]**. On the ninth pass, the ninth largest value sinks to **a[1]**. This leaves the smallest value in **a[0]**, so only nine passes of the array are needed to sort the array even though there are ten elements.

The sorting is performed by the nested **for** loop (lines 17–25). If a swap is necessary, it is performed by the three assignments

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

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

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

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

The chief virtue of the bubble sort is that it is easy to program. However, the bubble sort runs slowly. This becomes apparent when sorting large arrays. In the exercises, we will develop more efficient versions of the bubble sort. Far more efficient sorts than the bubble sort have been developed. We will investigate a few of these later in the text. More advanced courses investigate sorting and searching in greater depth.

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

We now consider a larger example. Computers are commonly used to compile and analyze the results of surveys and opinion polls. Figure 6.16 uses array **response** initialized with 99 (represented by symbolic constant **SIZE** in line 5) 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.

```
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  void mean( const int [] );
8  void median( int [] );
9  void mode( int [], const int [] ) ;
10 void bubbleSort( int [] );
11 void printArray( const int [] );
12
13 int main()
14 {
15    int frequency[ 10 ] = { 0 };
16    int response[ SIZE ] =
17       { 6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
18         7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
19         6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
20         7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
21         6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
22         7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
23         5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
24         7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
25         7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
26         4, 5, 6, 1, 6, 5, 7, 8, 7 };
27
28    mean( response );
29    median( response );
30    mode( frequency, response );
31    return 0;
32 }
33
34 void mean( const int answer[] )
35 {
36    int j, total = 0;
37
38    printf( "%s\n%s\n%s\n", "********", "  Mean", "********" );
39
40    for ( j = 0; j <= SIZE - 1; j++ )
41       total += answer[  j  ];
42
43    printf( "The mean is the average value of the data\n"
44            "items. The mean is equal to the total of\n"
```

Fig. 6.16 Survey data analysis program (part 1 of 3).

```
45             "all the data items divided by the number\n"
46             "of data items ( %d ). The mean value for\n"
47             "this run is: %d / %d = %.4f\n\n",
48             SIZE, total, SIZE, (  double  ) total / SIZE );
49  }
50
51  void median( int answer[] )
52  {
53     printf( "\n%s\n%s\n%s\n%s",
54             "********", " Median", "********",
55             "The unsorted array of responses is" );
56
57     printArray( answer );
58     bubbleSort( answer );
59     printf( "\n\nThe sorted array is" );
60     printArray( answer );
61     printf( "\n\nThe median is element %d of\n"
62             "the sorted %d element array.\n"
63             "For this run the median is %d\n\n",
64             SIZE / 2, SIZE, answer[ SIZE / 2 ] );
65  }
66
67  void mode( int freq[], const int answer[] )
68  {
69     int rating, j, h, largest = 0, modeValue = 0;
70
71     printf( "\n%s\n%s\n%s\n",
72             "********", "  Mode", "********" );
73
74     for ( rating = 1; rating <= 9; rating++ )
75        freq[ rating ] = 0;
76
77     for ( j = 0; j <= SIZE - 1; j++ )
78        ++freq[ answer[ j ] ];
79
80     printf( "%s%11s%19s\n\n%54s\n%54s\n\n",
81             "Response", "Frequency", "Histogram",
82             "1    1    2    2", "5    0    5    0    5" );
83
84     for ( rating = 1; rating <= 9; rating++ ) {
85        printf( "%8d%11d          ", rating, freq[ rating ] );
86
87        if ( freq[ rating ] > largest ) {
88           largest = freq[ rating ];
89           modeValue = rating;
90        }
91
92        for ( h = 1; h <= freq[ rating ]; h++ )
93           printf( "*" );
94
95        printf( "\n" );
96     }
97
```

Fig. 6.16 Survey data analysis program (part 2 of 3).

```
98     printf( "The mode is the most frequent value.\n"
99             "For this run the mode is %d which occurred"
100            " %d times.\n", modeValue, largest );
101 }
102
103 void bubbleSort( int a[] )
104 {
105    int pass, j, hold;
106
107    for ( pass = 1; pass <= SIZE - 1; pass++ )
108
109       for ( j = 0; j <= SIZE - 2; j++ )
110
111          if ( a[ j ] > a[ j + 1 ] ) {
112             hold = a[ j ];
113             a[ j ] = a[ j + 1 ];
114             a[ j + 1 ] = hold;
115          }
116 }
117
118 void printArray( const int a[] )
119 {
120    int j;
121
122    for ( j = 0; j <= SIZE - 1; j++ ) {
123
124       if ( j % 20 == 0 )
125          printf( "\n" );
126
127       printf( "%2d", a[ j ] );
128    }
129 }
```

Fig. 6.16 Survey data analysis program (part 3 of 3).

The mean is the arithmetic average of the 99 values. Function **mean** (line 34) computes the mean by totaling the 99 elements and dividing the result by 99.

The median is the "middle value." Function **median** (line 51) determines the median by calling function **bubbleSort** (defined in line 103) to sort the array of responses into ascending order and picking the middle element, **answer[SIZE / 2]**, of the sorted array. Note that 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 118) is called to output the **response** array.

The mode is the value that occurs most frequently among the 99 responses. Function **mode** (line 67) determines the mode by counting the number of responses of each type, then selecting the value with the greatest count. This version of function **mode** does not handle a tie (see Exercise 6.14). Function **mode** also produces a histogram to aid in determining the mode graphically. 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.

```
********
 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
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 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.

6.8 Searching Arrays

Often, a programmer will be working with large amounts of data stored in arrays. It may be necessary to determine whether an array contains a value that matches a certain *key value*.

The process of finding a particular element of an array is called *searching*. In this section we discuss two searching techniques—the simple *linear search* technique and the more efficient *binary search* technique. Exercises 6.34 and 6.35 at the end of this chapter ask you to implement recursive versions of the linear search and the binary search.

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 is just as likely that the value will be found in the first element as 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  int linearSearch( const int [], int, int );
7
8  int main()
9  {
10    int a[ SIZE ], x, searchKey, element;
11
12    for ( x = 0; x <= SIZE - 1; x++ )  /* create data */
13       a[ x ] = 2 * x;
14
15    printf( "Enter integer search key:\n" );
16    scanf( "%d", &searchKey );
17    element = linearSearch( a, searchKey, SIZE );
18
19    if ( element != -1 )
20       printf( "Found value in element %d\n", element );
21    else
22       printf( "Value not found\n" );
23
24    return 0;
25 }
26
27 int linearSearch( const int array[], int key, int size )
28 {
29    int n;
30
31    for ( n = 0; n <= size - 1; ++n )
32       if ( array[ n ] == key )
33          return n;
34
35    return -1;
36 }
```

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

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

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

Fig. 6.18 Linear search of an array (part 2 of 2).

The linear searching method works well for small arrays or for 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 one half of the elements in a sorted array after each comparison. The algorithm locates the middle element of the array and compares it to the search key. If they 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. 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 1024 elements will take 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. 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 the elements in the array. For a one billion element array, this is a difference between an average of 500 million comparisons and a maximum of 30 comparisons! The maximum comparisons for any array can be determined by finding the first power of 2 greater than the number of elements in the array.

Figure 6.19 presents the iterative version of function **`binarySearch`**. The function (defined in line 32) receives four arguments—an integer array **`b`**, an integer **`searchKey`**, the **`low`** array subscript and the **`high`** array subscript. If the search key does not match the middle element of a subarray, the **`low`** subscript or **`high`** subscript is modified so a smaller subarray can be searched. If the search key is less than the middle element, the **`high`** subscript is set to **`middle - 1`** and the search is continued on the elements from **`low`** to **`middle - 1`**. If the search key is greater than the middle element, the **`low`** subscript is set to **`middle + 1`** and the search is continued on the elements from **`middle + 1`** to **`high`**. The program uses an array of 15 elements. The first power of 2 greater than the number of elements in this array is 16 (2^4), so a maximum of 4 comparisons are required to find the search key. The program uses function **`printHeader`** (line 54) to output the array subscripts and function **`printRow`** (line 73) 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 an array */
3  #include <stdio.h>
4  #define SIZE 15
5
6  int binarySearch( const int [], int, int, int );
7  void printHeader( void );
8  void printRow( const int [], int, int, int );
9
10 int main()
11 {
12    int a[ SIZE ], i, key, result;
13
14    for ( i = 0; i <= SIZE - 1; i++ )
15       a[ i ] = 2 * i;
16
17    printf( "Enter a number between 0 and 28: " );
18    scanf( "%d", &key );
19
20    printHeader();
21    result = binarySearch( a, key, 0, SIZE - 1 );
22
23    if ( result != -1 )
24       printf( "\n%d found in array element %d\n",
25               key, result );
26    else
27       printf( "\n%d not found\n", key );
28
29    return 0;
30 }
31
32 int binarySearch( const int b[], int searchKey,
33                   int low, int high )
34 {
35    int middle;
36
37    while ( low <= high ) {
38       middle = ( low + high ) / 2;
39
40       printRow( b, low, middle, high );
41
42       if ( searchKey == b[ middle ] )
43          return middle;
44       else if ( searchKey < b[ middle ] )
45          high = middle - 1;
46       else
47          low = middle + 1;
48    }
49
```

Fig. 6.19 Binary search of a sorted array.

```
50       return -1;   /* searchKey not found */
51    }
52
53    /* Print a header for the output */
54    void printHeader( void )
55    {
56       int i;
57
58       printf( "\nSubscripts:\n" );
59
60       for ( i = 0; i <= SIZE - 1; i++ )
61          printf( "%3d ", i );
62
63       printf( "\n" );
64
65       for ( i = 1; i <= 4 * SIZE; i++ )
66          printf( "-" );
67
68       printf( "\n" );
69    }
70
71    /* Print one row of output showing the current
72       part of the array being processed. */
73    void printRow( const int b[], int low, int mid, int high )
74    {
75       int i;
76
77       for ( i = 0; i <= SIZE - 1; i++ )
78          if ( i < low || i > high )
79             printf( "    " );
80          else if ( i == mid )
81             printf( "%3d*", b[ i ] );  /* mark middle value */
82          else
83             printf( "%3d ", b[ i ] );
84
85       printf( "\n" );
86    }
```

```
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.

```
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.

6.9 Multiple-Subscripted Arrays

Arrays in C can have multiple subscripts. A common use of multiple-subscripted 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.* Note that multiple-subscripted arrays can have more than two subscripts. The ANSI standard states that an ANSI C system must support at least 12 array subscripts.

Figure 6.20 illustrates a double-subscripted array, **a**. The array contains three rows and four columns, so it is 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.*

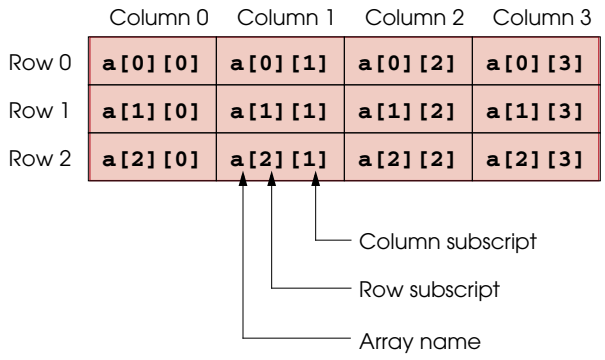


Fig. 6.20 A double-subscripted array with three rows and four columns.

Every element in array **a** is identified in Fig. 6.20 by an element name of the form **a [i] [j]**; **a** is the name of the array, and **i** and **j** are the subscripts that uniquely identify each element in **a**. Notice that 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 **a[x , y]** *instead of* **a[x][y]**.

A multiple-subscripted array can be initialized in its declaration much like a single subscripted array. For example, a double-subscripted array **b[2][2]** could be declared and initialized with

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

The values are grouped by row in braces. So, **1** and **2** initialize **b[0][0]** and **b[0][1]**, and **3** and **4** initialize **b[1][0]** and **b[1][1]**. If there are not enough initializers for a given row, the remaining elements of that row are initialized to **0**. Thus, the declaration

```
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 initializing double-subscripted arrays in declarations. The program declares three arrays of two rows and three columns (six elements each). The declaration of **array1** (line 9) provides six initializers in two sublists. The first sublist initializes the first row of the array to the values 1, 2 and 3; and the second sublist initializes the second row of the array to the values 4, 5 and 6. If the braces around each sublist are removed from the **array1** initializer list, the compiler initializes the elements of the first row followed by the elements of the second row. The declaration of **array2** (line 10) 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 declaration of **array3** (line 11) provides three initializers in two sublists. The sublist for the first row explicitly initializes the first two elements of the first row to 1 and 2. The third element is initialized to zero. The sublist for the second row explicitly initializes the first element to 4. The last two elements are initialized to zero.

```
1  /* Fig. 6.21: fig06_21.c
2     Initializing multidimensional arrays */
3  #include <stdio.h>
4
5  void printArray( const int [][ 3 ] );
6
7  int main()
8  {
9     int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } },
10        array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 },
11        array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };
```

Fig. 6.21 Initializing multidimensional arrays (part 1 of 2).

```
12
13    printf( "Values in array1 by row are:\n" );
14    printArray( array1 );
15
16    printf( "Values in array2 by row are:\n" );
17    printArray( array2 );
18
19    printf( "Values in array3 by row are:\n" );
20    printArray( array3 );
21
22    return 0;
23 }
24
25 void printArray( const int a[][ 3 ] )
26 {
27    int i, j;
28
29    for ( i = 0; i <= 1; i++ ) {
30
31       for ( j = 0; j <= 2; j++ )
32          printf( "%d ", a[ i ][ j ] );
33
34       printf( "\n" );
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).

The program calls function **printArray** (defined in line 25) to output each array's elements. Notice that the function definition specifies the array parameter as **const int a[][3]**. When we receive a single-subscripted array as an argument to a function, the array brackets are empty in the function's parameter list. 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. 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 exactly how many elements are in each row so 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 in memory to get to the second row (row 1). Then, the compiler accesses the third element of that row (element 2).

Many common array manipulations use **for** repetition structures. For example, the following structure 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** (**0** is the first row and **1** is the second row). The **for** loop varies only the second subscript (i.e., the column subscript). The preceding **for** structure is equivalent to the assignment statements:

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

The following nested **for** structure 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** structure totals the elements of the array one row at a time. The outer **for** structure begins by setting **row** (i.e., the row subscript) to **0** so the elements of the first row may be totaled by the inner **for** structure. The outer **for** structure increments **row** to **1**, so the elements of the second row can be totaled. Then, the outer **for** structure increments **row** to **2**, so the elements of the third row can be totaled. The result is printed when the nested **for** structure terminates.

Figure 6.22 performs several other common array manipulations on 3-by-4 array **studentGrades** using **for** structures. 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** (line 35) determines the lowest grade of any student for the semester. Function **maximum** (line 49) determines the highest grade of any student for the semester. Function **average** (line 63) determines a particular student's semester average. Function **printArray** (line 74) 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  int minimum( const int [][ EXAMS ], int, int );
8  int maximum( const int [][ EXAMS ], int, int );
```

Fig. 6.22 Example of using double-subscripted arrays (part 1 of 3).

```
 9  double average( const int [], int );
10  void printArray( const int [][ EXAMS ], int, int );
11
12  int main()
13  {
14     int student;
15     const int studentGrades[ STUDENTS ][ EXAMS ] =
16        { { 77, 68, 86, 73 },
17          { 96, 87, 89, 78 },
18          { 70, 90, 86, 81 } };
19
20     printf( "The array is:\n" );
21     printArray( studentGrades, STUDENTS, EXAMS );
22     printf( "\n\nLowest grade: %d\nHighest grade: %d\n",
23             minimum( studentGrades, STUDENTS, EXAMS ),
24             maximum( studentGrades, STUDENTS, EXAMS ) );
25
26     for ( student = 0; student <= STUDENTS - 1; student++ )
27        printf( "The average grade for student %d is %.2f\n",
28                student,
29                average( studentGrades[ student ], EXAMS ) );
30
31     return 0;
32  }
33
34  /* Find the minimum grade */
35  int minimum( const int grades[][ EXAMS ],
36               int pupils, int tests )
37  {
38     int i, j, lowGrade = 100;
39
40     for ( i = 0; i <= pupils - 1; i++ )
41        for ( j = 0; j <= tests - 1; j++ )
42           if ( grades[ i ][ j ] < lowGrade )
43              lowGrade = grades[ i ][ j ];
44
45     return lowGrade;
46  }
47
48  /* Find the maximum grade */
49  int maximum( const int grades[][ EXAMS ],
50               int pupils, int tests )
51  {
52     int i, j, highGrade = 0;
53
54     for ( i = 0; i <= pupils - 1; i++ )
55        for ( j = 0; j <= tests - 1; j++ )
56           if ( grades[ i ][ j ] > highGrade )
57              highGrade = grades[ i ][ j ];
58
59     return highGrade;
60  }
61
```

Fig. 6.22 Example of using double-subscripted arrays (part 2 of 3).

```
62  /* Determine the average grade for a particular exam */
63  double average( const int setOfGrades[], int tests )
64  {
65     int i, total = 0;
66
67     for ( i = 0; i <= tests - 1; i++ )
68        total += setOfGrades[ i ];
69
70     return ( double ) total / tests;
71  }
72
73  /* Print the array */
74  void printArray( const int grades[][ EXAMS ],
75                   int pupils, int tests )
76  {
77     int i, j;
78
79     printf( "                  [0]  [1]  [2]  [3]" );
80
81     for ( i = 0; i <= pupils - 1; i++ ) {
82        printf( "\nstudentGrades[%d] ", i );
83
84        for ( j = 0; j <= tests - 1; j++ )
85           printf( "%-5d", grades[ i ][ j ] );
86     }
87  }
```

```
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 Example of using double-subscripted arrays (part 3 of 3).

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

```
for ( i = 0; i <= pupils - 1; i++ )
   for ( j = 0; j <= tests - 1; j++ )
      if ( grades[ i ][ j ] < lowGrade )
         lowGrade = grades[ i ][ j ];
```

The outer **for** structure begins by setting **i** (i.e., the row subscript) to **0** so the elements of the first row can be compared to variable **lowGrade** in the body of the inner **for** struc-

ture. The inner **for** structure 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** structure then increments the row subscript to **1**. The elements of the second row are compared to variable **lowGrade**. The outer **for** structure 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 63) 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. 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.

SUMMARY

- C stores lists of values in arrays. An array is a group of related memory locations. These locations are related by the fact that they all have the same name and the same type. To refer to a particular location or element within the array, we specify the name of the array and the subscript.
- A subscript may be an integer or an integer expression. If a program uses an expression as a subscript, then the expression is evaluated to determine the particular element of the array.
- It is important to note the difference when referring to the seventh element of the array as opposed to array element seven. The seventh element has a subscript of **6**, while array element seven has a subscript of **7** (actually the eighth element of the array). This is a source of "off-by-one" errors.
- Arrays occupy space in memory. To reserve 100 elements for integer array **b** and 27 elements for integer array **x**, the programmer writes

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

- An array of type **char** can be used to store a character string.
- The elements of an array can be initialized three ways: by declaration, by assignment and by input.
- If there are fewer initializers than elements in the array, C initializes the remaining elements to zero.
- C does not prevent referencing elements beyond the bounds of an array.
- A character array can be initialized using a string literal.
- All strings in C end with the null character. The character constant representation of the null character is **'\0'**.
- Character arrays can be initialized with character constants in an initializer list.
- Individual characters in a string stored in an array can be accessed directly using array subscript notation.
- A string can be input directly into a character array from the keyboard using **scanf** and the conversion specification **%s**.
- A character array representing a string can be output with **printf** and the **%s** conversion specifier.

- Apply **`static`** to a local array declaration so the array is not created each time the function is called and the array is not destroyed each time the function exits.
- Arrays that are declared **`static`** are automatically initialized once at compile time. If the programmer does not explicitly initialize a **`static`** array, it is initialized to zero by the compiler.
- To pass an array to a function, the name of the array is passed. To pass a single element of an array to a function, simply pass the name of the array followed by the subscript (contained in square brackets) of the particular element.
- C passes arrays to functions using simulated call by reference—the called functions can modify the element values in the callers' original arrays. The name of the array is actually 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.
- To receive an array argument, the function's parameter list must specify that an array will be received. The size of the array is not required in the array brackets.
- The **`%p`** conversion specification normally outputs addresses as hexadecimal numbers.
- C provides the special type qualifier **`const`** to prevent modification of array values in a function. When an array parameter is preceded by the **`const`** qualifier, the elements of the array 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.
- An array can be sorted using the bubble sort technique. Several passes of the array are made. On each pass, successive pairs of elements are compared. If a pair is in order (or if the values are identical), it is left as is. If a pair is out of order, the values are swapped. For small arrays, the bubble sort is acceptable, but for larger arrays it is inefficient compared to other more sophisticated sorting algorithms.
- The linear search compares each element of the array with the search key. Since the array is not in any particular order, it is just as likely that the value will be found in the first element as the last. On average, therefore, the program will have to compare the search key with half the elements of the array. The linear searching method works well for small arrays or for unsorted arrays.
- The binary search algorithm eliminates one half of the elements in a sorted array after each comparison. The algorithm locates the middle element of the array and compares it to the search key. If they 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.
- In a worst case scenario, searching an array of 1024 elements will take only 10 comparisons using a binary search. 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 key.
- Arrays may be used to represent tables of values consisting of information arranged in rows and columns. To identify a particular element of a table, two subscripts are specified: The first (by convention) identifies the row in which the element is contained and the second (by convention) identifies the column in which the element is contained. Tables or arrays that require two subscripts to identify a particular element are called double-subscripted arrays.
- The C standard states that a system must support at least 12 array subscripts.
- A multiple-subscripted array can be initialized in its declaration using an initializer list.
- When we receive a single-subscripted array as an argument to a function, the array brackets are empty in the function's parameter list. 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.
- To pass one row of a double-subscripted array to a function that receives a single-subscripted array, simply pass the name of the array followed by the first subscript.

TERMINOLOGY

a[i]
a[i][j]
array
array initializer list
bar chart
bounds checking
bubble sort
column subscript
const qualifier
declare an array
#define preprocessor directive
double precision
double-subscripted array
element of an array
expression as a subscript
histogram
initialize an array
linear search
m-by-*n* array
mean
median
mode
multiple-subscripted array
name of an array
null character **'\0'**
off-by-one error
pass-by-reference
passing arrays to functions
%p conversion specification
position number
replacement text
row subscript
scalability
scalar
scalar quantity
search key
searching an array
single-subscripted array
sinking sort
sorting
sorting pass
sorting the elements of an array
square brackets
string
subscript
survey data analysis
symbolic constant
table of values
tabular format
temporary area for exchange of values
totaling the elements of an array
triple-subscripted array
value of an element
walk off an array
zeroth element

COMMON PROGRAMMING ERRORS

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

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

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

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.

6.6 Referring to an element outside the array bounds.

6.7 Not providing **scanf** with a character array large enough to store a string typed at the keyboard can result in loss of data in a program and other run-time errors.

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

6.9 Referencing a double-subscripted array element as **a[x, y]** instead of **a[x][y]**.

GOOD PROGRAMMING PRACTICES

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

6.2 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.

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

6.4 Mention the high array subscript in a **`for`** structure to help eliminate off-by-one errors.

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

6.6 Some programmers include variable names in function prototypes to make programs clearer. The compiler ignores these names.

PERFORMANCE TIPS

6.1 Sometimes performance considerations far outweigh clarity considerations.

6.2 The (normally serious) effects of referencing elements outside the array bounds are system dependent.

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

6.4 Passing arrays simulated call by reference makes sense for performance reasons. If arrays were passed call by value, a copy of each element would be passed. For large, frequently passed arrays, this would be time consuming and would consume considerable storage for the copies of the arrays.

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

SOFTWARE ENGINEERING OBSERVATIONS

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

6.2 It is possible to pass an array by value by using a simple trick we explain in Chapter 10.

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 is absolutely necessary.

SELF-REVIEW EXERCISES

6.1 Answer each of the following:

a) Lists and tables of values are stored in _____.

b) The elements of an array are related by the fact that they have the same _____ and _____.

c) The number used to refer to a particular element of an array is called its _____.

d) A _____ should be used to declare the size of an array because it makes the program more scalable.

e) The process of placing the elements of an array in order is called _____ the array.

f) Determining if an array contains a certain key value is called _____ the array.

g) An array that uses two subscripts is referred to as a _____ array.

6.2 State whether the following are *true* or *false*. If the answer is *false*, explain why.

a) An array can store many different types of values.
b) An array subscript can be of data type **double**.
c) If there are fewer initializers in an initializer list than the number of elements in the array, C automatically initializes the remaining elements to the last value in the list of initializers.
d) It is an error if an initializer list contains more initializers than there are elements in the array.
e) An individual array element that is passed to a function and modified in the called function will contain the modified value in the calling function.

6.3 Answer the following questions regarding an array called **fractions**.

a) Define a symbolic constant **SIZE** to be replaced with the replacement text 10.
b) Declare an array with **SIZE** elements of type **double** and initialize the elements to **0**.
c) Name the fourth element from the beginning of the array.
d) Refer to array element 4.
e) Assign the value **1.667** to array element nine.
f) Assign the value **3.333** to the seventh element of the array.
g) Print array elements 6 and 9 with two digits of precision to the right of the decimal point, and show the output that is actually displayed on the screen.
h) Print all the elements of the array using a **for** repetition structure. Assume the integer variable **x** has been defined as a control variable for the loop. Show the output.

6.4 Write statements to accomplish the following:

a) Declare **table** to be an integer array and to have 3 rows and 3 columns. Assume the symbolic constant **SIZE** has been defined to be 3.
b) How many elements does the array **table** contain? Print the total number of elements.
c) Use a **for** repetition structure to initialize each element of **table** to the sum of its subscripts. Assume the integer variables **x** and **y** are declared as control variables.
d) Print the values of each element of array **table**. Assume the array was initialized with the declaration:

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

6.5 Find the error in each of the following program segments and correct the error.

a)
```
#define SIZE 100;
```
b)
```
SIZE = 10;
```
c) Assume
```
int b[ 10 ] = { 0 }, i;
   for ( i = 0; i <= 10; i++ )
      b[ i ] = 1;
```
d)
```
#include <stdio.h>;
```
e) Assume
```
int a[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
   a[ 1, 1 ] = 5;
```
f)
```
#define VALUE = 120
```

ANSWERS TO SELF-REVIEW EXERCISES

6.1 a) Arrays. b) Name, type. c) Subscript. d) Symbolic constant. e) Sorting. f) Searching. g) Double-subscripted.

6.2

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

6.3 a) `#define SIZE 10`
b) `double fractions[ SIZE ] = { 0 };`
c) `fractions[ 3 ]`
d) `fractions[ 4 ]`
e) `fractions[ 9 ] = 1.667;`
f) `fractions[ 6 ] = 3.333;`
g) `printf( "%.2f %.2f\n", fractions[ 6 ], fractions[ 9 ] );`
h) *Output:* `3.33 1.67`.
i)
```
for ( x = 0; x <= SIZE - 1; x++ )
   printf( "fractions[%d] = %f\n", x, fractions[ x ] );
```

Output:

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

6.4 a) `int table[ SIZE ][ SIZE ];`
b) Nine elements. `printf( "%d\n", SIZE * SIZE );`
c)
```
for ( x = 0; x <= SIZE - 1; x++ )
   for ( y = 0; y <= SIZE - 1; y++ )
      table[ x ][ y ] = x + y;
```
d)
```
for ( x = 0; x <= SIZE - 1; x++ )
   for ( y = 0; y <= SIZE - 1; y++ )
      printf( "table[%d][%d] = %d\n", x, y, table[ x ][ y ] );
```

Output:

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

6.5 a) Error: Semicolon at end of `#define` preprocessor directive.
Correction: Eliminate semicolon.
b) Error: Assigning a value to a symbolic constant using an assignment statement.
Correction: Assign a value to the symbolic constant in a `#define` preprocessor directive without using the assignment operator as in `#define SIZE 10`.
c) Error: Referencing an array element outside the bounds of the array (`b[ 10 ]`).
Correction: Change the final value of the control variable to `9`.
d) Error: Semicolon at end of `#include` preprocessor directive.
Correction: Eliminate semicolon.
e) Error: Array subscripting done incorrectly.
Correction: Change the statement to `a[ 1 ][ 1 ] = 5;`

g) Error: Assigning a value to a symbolic constant using an assignment statement.
Correction: Assign a value to the symbolic constant in a **#define** preprocessor directive without using the assignment operator as in **#define VALUE 120**.

EXERCISES

6.6 Fill in the blanks in each of the following:

a) C stores lists of values in _____.
b) The elements of an array are related by the fact that they _____.
c) When referring to an array element, the position number contained within parentheses is called a _____.
d) The names of the five elements of array **p** are _____, _____, _____, _____ and _____.
e) The contents of a particular element of an array is called the _____ of that element.
f) Naming an array, stating its type and specifying the number of elements in the array is called _____ the array.
g) The process of placing the elements of an array into either ascending or descending order is called _____.
h) In a double-subscripted array, the first subscript (by convention) identifies the _____ of an element and the second subscript (by convention) identifies the _____ of an element.
i) An *m*-by-*n* array contains _____ rows, _____ columns and _____ elements.
j) The name of the element in row 3 and column 5 of array **d** is _____.

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

a) To refer to a particular location or element within an array, we specify the name of the array and the value of the particular element.
b) An array declaration reserves space for the array.
c) To indicate that 100 locations should be reserved for integer array **p**, the programmer writes the declaration

```
p[ 100 ];
```

d) A C program that initializes the elements of a 15-element array to zero must contain one **for** statement.
e) A C program that totals the elements of a double-subscripted array must contain nested **for** statements.
f) The mean, median and mode of the following set of values are 5, 6 and 7, respectively: 1, 2, 5, 6, 7, 7, 7.

6.8 Write statements to accomplish each of the following:

a) Display the value of the seventh element of character array **f**.
b) Input a value into element 4 of single-subscripted floating-point array **b**.
c) Initialize each of the 5 elements of single-subscripted integer array **g** to **8**.
d) Total the elements of floating-point array **c** of 100 elements.
e) Copy array **a** into the first portion of array **b**. Assume **double a[11], b[34];**
f) Determine and print the smallest and largest values contained in 99-element floating-point array **w**.

6.9 Consider a 2-by-5 integer array **t**.

a) Write a declaration for **t**.
b) How many rows does **t** have?
c) How many columns does **t** have?

d) How many elements does **t** have?
e) Write the names of all the elements in the second row of **t**.
f) Write the names of all the elements in the third column of **t**.
g) Write a single statement that sets the element of **t** in row 1 and column 2 to zero.
h) Write a series of statements that initialize each element of **t** to zero. Do not use a repetition structure.
i) Write a nested **for** structure that initializes each element of **t** to zero.
j) Write a statement that inputs the values for the elements of **t** from the terminal.
k) Write a series of statements that determine and print the smallest value in array **t**.
l) Write a statement that displays the elements of the first row of **t**.
m) Write a statement that totals the elements of the fourth column of **t**.
n) Write a series of statements that print the array **t** in neat, tabular format. List the column subscripts as headings across the top and list the row subscripts at the left of each row.

6.10 Use a single-subscripted array to solve the following problem. A company pays its salespeople on a commission basis. The salespeople receive $200 per week plus 9 percent of their gross sales for that week. For example, a salesperson who grosses $3000 in sales in a week receives $200 plus 9 percent of $3000, or a total of $470. Write a C program (using an array of counters) that determines how many of the salespeople earned salaries in each of the following ranges (assume that each salesperson's salary is truncated to an integer amount):

1. $200–$299
2. $300–$399
3. $400–$499
4. $500–$599
5. $600–$699
6. $700–$799
7. $800–$899
8. $900–$999
9. $1000 and over

6.11 The bubble sort presented in Fig. 6.15 is inefficient for large arrays. Make the following simple modifications to improve the performance of the bubble sort.

a) After the first pass, the largest number is guaranteed to be in the highest-numbered element of the array; after the second pass, the two highest numbers are "in place," and so on. Instead of making nine comparisons on every pass, modify the bubble sort to make eight comparisons on the second pass, seven on the third pass and so on.
b) The data in the array may already be in the proper order or near-proper order, so why make nine passes if fewer will suffice? Modify the sort to check at the end of each pass if any swaps have been made. If none has been made, then the data must already be in the proper order, so the program should terminate. If swaps have been made, then at least one more pass is needed.

6.12 Write single statements that perform each of the following single-subscripted array operations:

a) Initialize the 10 elements of integer array **counts** to zeros.
b) Add 1 to each of the 15 elements of integer array **bonus**.
c) Read the 12 values of floating-point array **monthlyTemperatures** from the keyboard.
d) Print the 5 values of integer array **bestScores** in column format.

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

a) Assume: **char str[5];**

```
scanf( "%s", str );      /* User types hello */
```

b) Assume: `int a[ 3 ];`
`printf( "$d %d %d\n", a[ 1 ], a[ 2 ], a[ 3 ] );`
c) `double f[ 3 ] = { 1.1, 10.01, 100.001, 1000.0001 };`
d) Assume: `double d[ 2 ][ 10 ];`
`d[ 1, 9 ] = 2.345;`

6.14 Modify the program of Fig. 6.16 so function **mode** is capable of handling a tie for the mode value. Also modify function **median** so the two middle elements are averaged in an array with an even number of elements.

6.15 Use a single-subscripted array to solve the following problem. Read in 20 numbers, each of which is between 10 and 100, inclusive. As each number is read, print it only if it is not a duplicate of a number already read. Provide for the "worst case" in which all 20 numbers are different. Use the smallest possible array to solve this problem.

6.16 Label the elements of 3-by-5 double-subscripted array **sales** to indicate the order in which they are set to zero by the following program segment:

```
for ( row = 0; row <= 2; row++ )
   for ( column = 0; column <= 4; column++ )
      sales[ row ][ column ] = 0;
```

6.17 What does the following program do?

```
1  /* ex06_17.c */
2  #include <stdio.h>
3  #define SIZE 10
4
5  int whatIsThis( const int [], int );
6
7  int main()
8  {
9     int x, a[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
10
11    x = whatIsThis( a, SIZE );
12    printf( "Result is %d\n", x );
13    return 0;
14 }
15
16 int whatIsThis( const int b[], int p )
17 {
18    if ( p == 1 )
19       return b[ 0 ];
20    else
21       return b[ p - 1 ] + whatIsThis( b, p - 1 );
22 }
```

6.18 What does the following program do?

```
1  /* ex06_18.c */
2  #include <stdio.h>
3  #define SIZE 10
4
5  void someFunction( const int [], int );
```

```
6
7  int main()
8  {
9     int a[ SIZE ] = { 8, 3, 1, 2, 6, 0, 9, 7, 4, 5 };
10
11    printf( "Answer is:\n" );
12    someFunction( a, SIZE );
13    printf( "\n" );
14    return 0;
15 }
16
17 void someFunction( const int b[], int size )
18 {
19    if ( size > 0 ) {
20       someFunction( &b[ 1 ], size - 1 );
21       printf( "%d  ", b[ 0 ] );
22    }
23 }
```

6.19 Write a program that simulates the rolling of two dice. The program should use **rand** to roll the first die, and should use **rand** again to roll the second die. The sum of the two values should then be calculated. (*Note:* Since each die can show an integer value from 1 to 6, then the sum of the two values will vary from 2 to 12 with 7 being the most frequent sum and 2 and 12 being the least frequent sums.) Figure 6.23 shows the 36 possible combinations of the two dice. Your program should roll the two dice 36,000 times. Use a single-subscripted array to tally the numbers of times each possible sum appears. Print the results in a tabular format. Also, determine if the totals are reasonable; i.e., there are six ways to roll a 7, so approximately one sixth of all the rolls should be 7.

6.20 Write a program that runs 1000 games of craps and answers each of the following questions:

a) How many games are won on the first roll, second roll, …, twentieth roll and after the twentieth roll?

b) How many games are lost on the first roll, second roll, …, twentieth roll and after the twentieth roll?

c) What are the chances of winning at craps? (*Note:* You should discover that craps is one of the fairest casino games. What do you suppose this means?)

d) What is the average length of a game of craps?

e) Do the chances of winning improve with the length of the game?

6.21 (*Airline Reservations System*) A small airline has just purchased a computer for its new automated reservations system. The president has asked you to program the new system. You are to write a program to assign seats on each flight of the airline's only plane (capacity: 10 seats).

Your program should display the following menu of alternatives:

```
Please type 1 for "first class"
Please type 2 for "economy"
```

If the person types 1, then your program should assign a seat in the first class section (seats 1-5). If the person types 2, then your program should assign a seat in the economy section (seats 6-10). Your program should then print a boarding pass indicating the person's seat number and whether it is in the first class or economy section of the plane.

Use a single-subscripted array to represent the seating chart of the plane. Initialize all the elements of the array to 0 to indicate that all seats are empty. As each seat is assigned, set the corresponding elements of the array to 1 to indicate that the seat is no longer available.

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

Fig. 6.23 The 36 possible outcomes of rolling two dice.

Your program should, of course, never assign a seat that has already been assigned. When the first class section is full, your program should ask the person if it is acceptable to be placed in the economy section (and vice versa). If yes, then make the appropriate seat assignment. If no, then print the message **`"Next flight leaves in 3 hours."`**

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

1. The salesperson number
2. The product number
3. The total dollar value of that product sold that day

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

6.23 (*Turtle Graphics*) The Logo language, which is particularly popular among personal computer users, made the concept of *turtle graphics* famous. Imagine a mechanical turtle that walks around the room under the control of a C program. The turtle holds a pen in one of two positions, up or down. While the pen is down, the turtle traces out shapes as it moves; while the pen is up, the turtle moves about freely without writing anything. In this problem you will simulate the operation of the turtle and create a computerized sketchpad as well.

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

Command	Meaning
1	Pen up
2	Pen down
3	Turn right
4	Turn left
5,10	Move forward 10 spaces (or a number other than 10)
6	Print the 20-by-20 array
9	End of data (sentinel)

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

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

As the turtle moves with the pen down, set the appropriate elements of array **floor** to **1**s. When the **6** command (print) is given, wherever there is a 1 in the array, display an asterisk, or some other character you choose. Wherever there is a zero, display a blank. Write a program to implement the turtle graphics capabilities discussed here. Write several turtle graphics programs to draw interesting shapes. Add other commands to increase the power of your turtle graphics language.

6.24 (*Knight's Tour*) One of the more interesting puzzlers for chess buffs is the Knight's Tour problem, originally proposed by the mathematician Euler. The question is this: Can the chess piece called the knight move around an empty chessboard and touch each of the 64 squares once and only once? We study this intriguing problem in depth here.

The knight makes L-shaped moves (over two in one direction and then over one in a perpendicular direction). Thus, from a square in the middle of an empty chessboard, the knight can make eight different moves (numbered 0 through 7) as shown in Fig. 6.24.

a) Draw an 8-by-8 chessboard on a sheet of paper and attempt a Knight's Tour by hand. Put a **1** in the first square you move to, a **2** in the second square, a **3** in the third, etc. Before starting the tour, estimate how far you think you will get, remembering that a full tour consists of 64 moves. How far did you get? Were you close to the estimate?

b) Now let us develop a program that will move the knight around a chessboard. The board itself is represented by an 8-by-8 double-subscripted array **board**. Each of the squares is initialized to zero. We describe each of the eight possible moves in terms of both their horizontal and vertical components. For example, a move of type 0 as shown in Fig. 6.24 consists of moving two squares horizontally to the right and one square vertically upward. Move 2 consists of moving one square horizontally to the left and two squares vertically upward. Horizontal moves to the left and vertical moves upward are indicated with negative numbers. The eight moves may be described by two single-subscripted arrays, **horizontal** and **vertical**, as follows:

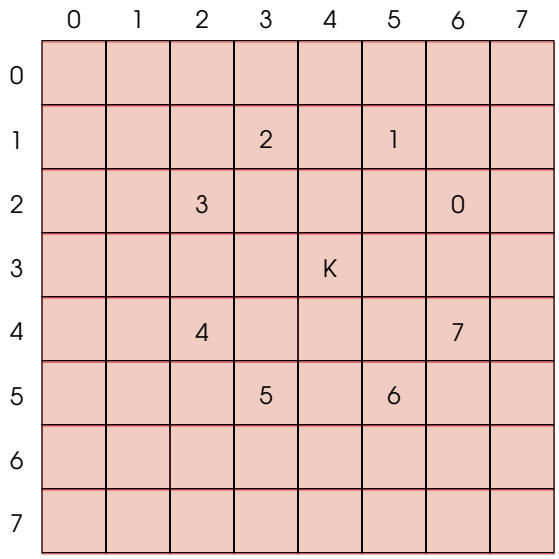


Fig. 6.24 The eight possible moves of the knight.

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

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

Let the variables **currentRow** and **currentColumn** indicate the row and column of the knight's current position on the board. To make a move of type **moveNumber**, where **moveNumber** is between 0 and 7, your program uses the statements

```
currentRow += vertical[ moveNumber ];
currentColumn += horizontal[ moveNumber ];
```

Keep a counter that varies from **1** to **64**. Record the latest count in each square the knight moves to. Remember to test each potential move to see if the knight has already visited that square. And, of course, test every potential move to make sure that the knight does not land off the chessboard. Now write a program to move the knight around the chessboard. Run the program. How many moves did the knight make?

c) After attempting to write and run a Knight's Tour program, you have probably developed some valuable insights. We will use these to develop a *heuristic* (or strategy) for moving the knight. Heuristics do not guarantee success, but a carefully developed heuristic greatly improves the chance of success. You may have observed that the outer squares are in some sense more troublesome than the squares nearer the center of the board. In fact, the most troublesome, or inaccessible, squares are the four corners.

Intuition may suggest that you should attempt to move the knight to the most troublesome squares first and leave open those that are easiest to get to so that when the board gets congested near the end of the tour there will be a greater chance of success.

We may develop an "accessibility heuristic" by classifying each of the squares according to how accessible they are and always moving the knight to the square (within the knight's L-shaped moves, of course) that is most inaccessible. We label a double-subscripted array **`accessibility`** with numbers indicating from how many squares each particular square is accessible. On a blank chessboard, the center squares are therefore rated as **`8`**s, the corner squares are rated as **`2`**s, and the other squares have accessibility numbers of **`3`**, **`4`**, or **`6`** as follows:

```
2 3 4 4 4 4 3 2
3 4 6 6 6 6 4 3
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
3 4 6 6 6 6 4 3
2 3 4 4 4 4 3 2
```

Now write a version of the Knight's Tour program using the accessibility heuristic. At any time, the knight should move to the square with the lowest accessibility number. In case of a tie, the knight may move to any of the tied squares. Therefore, the tour may begin in any of the four corners. (*Note:* As the knight moves around the chessboard, your program should reduce the accessibility numbers as more and more squares become occupied. In this way, at any given time during the tour, each available square's accessibility number will remain equal to precisely the number of squares from which that square may be reached.) Run this version of your program. Did you get a full tour? Now modify the program to run 64 tours, one from each square of the chessboard. How many full tours did you get?

d) Write a version of the Knight's Tour program which, when encountering a tie between two or more squares, decides what square to choose by looking ahead to those squares reachable from the "tied" squares. Your program should move to the square for which the next move would arrive at a square with the lowest accessibility number.

6.25 (*Knight's Tour: Brute Force Approaches*) In Exercise 6.24 we developed a solution to the Knight's Tour problem. The approach used, called the "accessibility heuristic," generates many solutions and executes efficiently.

As computers continue increasing in power, we will be able to solve many problems with sheer computer power and relatively unsophisticated algorithms. Let us call this approach "brute force" problem solving.

a) Use random number generation to enable the knight to walk around the chess board (in its legitimate L-shaped moves, of course) at random. Your program should run one tour and print the final chessboard. How far did the knight get?

b) Most likely, the preceding program produced a relatively short tour. Now modify your program to attempt 1000 tours. Use a single-subscripted array to keep track of the number of tours of each length. When your program finishes attempting the 1000 tours, it should print this information in neat tabular format. What was the best result?

c) Most likely, the preceding program gave you some "respectable" tours but no full tours. Now "pull all the stops out" and simply let your program run until it produces a full tour. (*Caution:* This version of the program could run for hours on a powerful computer.) Once again, keep a table of the number of tours of each length and print this table when the first full tour is found. How many tours did your program attempt before producing a full tour? How much time did it take?

d) Compare the brute force version of the Knight's Tour with the accessibility heuristic version. Which required a more careful study of the problem? Which algorithm was more difficult to develop? Which required more computer power? Could we be certain (in advance) of obtaining a full tour with the accessibility heuristic approach? Could we be certain (in advance) of obtaining a full tour with the brute force approach? Argue the pros and cons of brute force problem solving in general.

6.26 (*Eight Queens*) Another puzzler for chess buffs is the Eight Queens problem. Simply stated: Is it possible to place eight queens on an empty chessboard so that no queen is "attacking" any other; that is, so that no two queens are in the same row, the same column, or along the same diagonal? Use the kind of thinking developed in Exercise 6.24 to formulate a heuristic for solving the Eight Queens problem. Run your program. (*Hint:* It is possible to assign a numeric value to each square of the chessboard indicating how many squares of an empty chessboard are "eliminated" once a queen is placed in that square. For example, each of the four corners would be assigned the value 22, as in Fig. 6.25.)

Once these "elimination numbers" are placed in all 64 squares, an appropriate heuristic might be: Place the next queen in the square with the smallest elimination number. Why is this strategy intuitively appealing?

6.27 (*Eight Queens: Brute Force Approaches*) In this problem you will develop several brute force approaches to solving the Eight Queens problem introduced in Exercise 6.26.

a) Solve the Eight Queens problem, using the random brute force technique developed in Problem 6.25.

b) Use an exhaustive technique (i.e., try all possible combinations of eight queens on the chessboard).

c) Why do you suppose the exhaustive brute force approach may not be appropriate for solving the Knight's Tour problem?

d) Compare and contrast the random brute force and exhaustive brute force approaches in general.

6.28 (*Duplicate elimination*) In Chapter 12, we explore the high-speed binary search tree data structure. One feature of a binary search tree is that duplicate values are discarded when insertions are made into the tree. This is referred to as duplicate elimination. Write a program that produces 20 random numbers between 1 and 20. The program should store all nonduplicate values in an array. Use the smallest possible array to accomplish this task.

Fig. 6.25 The 22 squares eliminated by placing a queen in the upper-left corner.

6.29 (*Knight's Tour: Closed Tour Test*) In the Knight's Tour, a full tour is when the knight makes 64 moves touching each square of the chess board once and only once. A closed tour occurs when the 64th move is one move away from the location in which the knight started the tour. Modify the Knight's Tour program you wrote in Exercise 6.24 to test for a closed tour if a full tour has occurred.

6.30 (*The Sieve of Eratosthenes*) A prime integer is any integer that can be divided evenly only by itself and 1. The Sieve of Eratosthenes is a method of finding prime numbers. It works as follows:

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

When this process is complete, the array elements that are still set to one indicate that the subscript is a prime number. These subscripts can then be printed. Write a program that uses an array of 1000 elements to determine and print the prime numbers between 1 and 999. Ignore element 0 of the array.

6.31 (*Bucket Sort*) A bucket sort begins with an single-subscripted array of positive integers to be sorted, and a double-subscripted array of integers with rows subscripted from 0 to 9 and columns subscripted from 0 to *n* - 1 where *n* is the number of values in the array to be sorted. Each row of the double-subscripted array is referred to as a bucket. Write a function **`bucketSort`** that takes an integer array and the array size as arguments.

The algorithm is as follows:

1) Loop through the single-subscripted array and place each of its values in a row of the bucket array based on its ones digit. For example, 97 is placed in row 7, 3 is placed in row 3 and 100 is placed in row 0.
2) Loop through the bucket array and copy the values back to the original array. The new order of the above values in the single-subscripted array is 100, 3 and 97.
3) Repeat this process for each subsequent digit position (tens, hundreds, thousands, etc.) and stop when the leftmost digit of the largest number has be processed.

On the second pass of the array, 100 is placed in row 0, 3 is placed in row 0 (it had only one digit) and 97 is placed in row 9. The order of the values in the single-subscripted array is 100, 3 and 97. On the third pass, 100 is placed in row 1, 3 is placed in row zero and 97 is placed in row zero (after 3). The bucket sort is guaranteed to have all the values properly sorted after processing the leftmost digit of the largest number. The bucket sort knows it is done when all the values are copied into row zero of the double-subscripted array.

Note that the double-subscripted array of buckets is ten times the size of the integer array being sorted. This sorting technique provides better performance than a bubble sort, but requires much larger storage capacity. Bubble sort requires only one additional memory location for the type of data being sorted. Bucket sort is an example of a space-time trade-off. It uses more memory, but performs better. This version of the bucket sort requires copying all the data back to the original array on each pass. Another possibility is to create a second double-subscripted bucket array and repeatedly move the data between the two bucket arrays until all the data is copied into row zero of one of the arrays. Row zero then contains the sorted array.

RECURSION EXERCISES

6.32 (*Selection Sort*) A selection sort searches an array looking for the smallest element in the array. When the smallest element is found, it is swapped with the first element of the array. The process

is then repeated for the subarray beginning with the second element of the array. Each pass of the array results in one element being placed in its proper location. This sort requires similar processing capabilities to the bubble sort—for an array of *n* elements, *n* - 1 passes must be made, and for each subarray, *n* - 1 comparisons must be made to find the smallest value. When the subarray being processed contains one element, the array is sorted. Write a recursive function **`selectionSort`** to perform this algorithm.

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

6.34 (*Linear Search)* Modify the program of Fig. 6.18 to use a recursive **`linearSearch`** function to perform the linear search of the array. The function should receive an integer array and the size of the array as arguments. If the search key is found, return the array subscript; otherwise, return -1.

6.35 (*Binary Search*) Modify the program of Fig. 6.19 to use a recursive **`binarySearch`** function to perform the binary search of the array. The function should receive an integer array and the starting subscript and ending subscript as arguments. If the search key is found, return the array subscript; otherwise, return -1.

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

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

6.38 (*Print a string backwards*) Write a recursive function **`stringReverse`** that takes a character array as an argument and returns nothing. The function should stop processing and return when the terminating null character of the string is encountered.

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