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
4.1 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 6, we discuss the notions of structures and classes—each capable of holding 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 15, we introduce dynamic data structures such as lists, queues, stacks, and trees that may grow and shrink as programs execute. The style of arrays we use in this chapter are C-style pointer-based arrays (we will study pointers in Chapter 5). Later in the text in Chapter 8 on “Operator Overloading” and in Chapter 20 on “The Standard Template Library,” we will cover arrays as full-fledged objects using the techniques of object-oriented programming. We will discover that these object-based arrays are safer and more versatile than the C-like, pointer-based arrays we discuss here in Chapter 4.

4.2 Arrays

An array is a consecutive group of memory locations that 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 4.1 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 \( c[1] \) (1 element from the beginning of the array), the seventh element of array \( c \) is referred to as \( c[6] \) (6 elements from the beginning of the array), and, in general, the \( i \)th element of array \( c \) is referred to as \( c[i - 1] \). Array names follow the same conventions as other variable names.

The position number contained within square brackets is more formally called a subscript (this number specifies the number of elements from the beginning of the array). A subscript must be an integer or an integer expression (using any integral type). If a program uses an expression as a subscript, then the expression is evaluated to determine the subscript. For example, if we assume that variable \( a \) is equal to 5 and that variable \( b \) is equal to 6, then the statement

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

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.

---

**Fig. 4.1** A 12-element array.
Let us examine array c in Fig. 4.1 more closely. The name of the entire array is c. Its 12 elements are named c[0], c[1], c[2], ..., c[11]. The value of 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

```
```

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 4.1**

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. Unfortunately, this is a source of “off-by-one” errors.

The brackets used to enclose the subscript of an array are actually an operator in C++. Brackets have the same level of precedence as parentheses. The chart in Fig. 4.2 shows the precedence and associativity of the operators introduced so far. They are shown top to bottom in decreasing order of precedence with their associativity and type.

### 4.3 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 compiler may reserve the appropriate amount of memory. To tell the compiler to reserve 12 elements for integer array c, use the declaration

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

Memory may be reserved for several arrays with a single declaration. The following declaration reserves 100 elements for the integer array b and 27 elements for the integer array x.

```
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 (a relationship C++ inherited from C) and the relationship between pointers and arrays are discussed in Chapter 5. After we introduce object-oriented programming, we will consider strings as full-fledged objects.

### 4.4 Examples Using Arrays

The program in Fig. 4.3 uses a for repetition structure to initialize the elements of a ten-element integer array n to zeros and prints the array in tabular format. The first output statement displays the column headings for the columns printed in the subsequent for structure. Remember that `setw` specifies the field width in which the next value is to be output.
### Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>() [ ]</td>
<td>left to right</td>
<td>highest</td>
</tr>
<tr>
<td>++ -- static_cast&lt;typename&gt;()</td>
<td>left to right</td>
<td>postfix</td>
</tr>
<tr>
<td>++ -- + - !</td>
<td>right to left</td>
<td>unary</td>
</tr>
<tr>
<td>* / %</td>
<td>left to right</td>
<td>multiplicative</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
<td>additive</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>left to right</td>
<td>insertion/extraction</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
<td>relational</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
<td>equality</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>left to right</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>right to left</td>
<td>conditional</td>
</tr>
<tr>
<td>= += -= *= /= %=</td>
<td>right to left</td>
<td>assignment</td>
</tr>
<tr>
<td>,</td>
<td>left to right</td>
<td>comma</td>
</tr>
</tbody>
</table>

---

**Fig. 4.2** Operator precedence and associativity.

---

```cpp
// Fig. 4.3: fig04_03.cpp
// initializing an array
#include <iostream>
#include <iomanip>
using std::cout;
using std::endl;
using std::setw;
#include <iomanip>
using std::setw;
int main()
{
  int i, n[10];
  for (i = 0; i < 10; i++) // initialize array
    n[i] = 0;
  cout << "Element" << setw(13) << "Value" << endl;
  for (i = 0; i < 10; i++) // print array
    cout << setw(7) << i << setw(13) << n[i] << endl;
  return 0;
}
```

**Fig. 4.3** Initializing the elements of an array to zeros (part 1 of 2).
The elements of an array can also be initialized in the array declaration by following the declaration with an equals sign and a comma-separated list (enclosed in braces) of initializers. The program in Fig. 4.4 initializes an integer array with 10 values and prints the array in tabular format.

If there are fewer initializers than elements in the array, the remaining elements are automatically initialized to zero. For example, the elements of the array \texttt{n} in Fig. 4.3 could have been initialized to zero with the declaration

\begin{verbatim}
int n[10] = { 0 };
\end{verbatim}

which explicitly initializes the first element to zero and implicitly initializes the remaining nine elements to zero, because there are fewer initializers than elements in the array. Remember that automatic arrays are not implicitly initialized to zero. The programmer must at least initialize the first element to zero for the remaining elements to be automatically zeroed. The method used in Fig. 4.3 can be performed repeatedly as a program executes.

The array declaration

\begin{verbatim}
int n[5] = { 32, 27, 64, 18, 95, 14 };
\end{verbatim}

would cause a syntax error, because there are 6 initializers and only 5 array elements.

**Common Programming Error 4.2**

Forgetting to initialize the elements of an array whose elements should be initialized is a logic error.

**Common Programming Error 4.3**

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,

\begin{verbatim}
int n[] = { 1, 2, 3, 4, 5 };
\end{verbatim}

would create a five-element array.

**Performance Tip 4.1**

If, instead of initializing an array with execution-time assignment statements, you initialize the array at compile time with an array initializer list, your program will execute faster.
The program in Fig. 4.5 initializes the elements of a 10-element array \texttt{s} to the integers \texttt{2, 4, 6, ..., 20} and prints the array in tabular format. These numbers are generated by multiplying each successive value of the loop counter by \texttt{2} and adding \texttt{2}.
const int arraySize = 10;

int main()
{
    const int arraySize = 10;
    int j, s[ arraySize ];

    for ( j = 0; j < arraySize; j++ )   // set the values
        s[ j ] = 2 + 2 * j;

    for ( j = 0; j < arraySize; j++ )   // print the values
        cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << endl;

    return 0;
}

---

Fig. 4.5 Generating values to be placed into elements of an array (part 2 of 2).

Line 14

    const int arraySize = 10;

uses the `const` qualifier to declare a so-called constant variable `arraySize` the value of which is 10. Constant variables must be initialized with a constant expression when they are declared and cannot be modified thereafter (Fig. 4.6 and Fig. 4.7). Constant variables are also called named constants, or read-only variables. Note that the term “constant variable” is an oxymoron—a contradiction in terms like “jumbo shrimp” or “freezer burn.” (Please send your favorite oxymorons to our email address listed in the Preface. Thanks!)

---

Fig. 4.6 Correctly initializing and using a constant variable.
```cpp
int main()
{
    const int x = 7;  // initialized constant variable
    cout << "The value of constant variable x is: " << x << endl;
    return 0;
}

The value of constant variable x is: 7
```

**Fig. 4.6**  Correctly initializing and using a constant variable.

```cpp
int main()
{
    const int x;  // Error: x must be initialized
    x = 7;        // Error: cannot modify a const variable
    return 0;
}
```

**Borland C++ command-line compiler error messages**

Fig04_07.cpp:
Error E2304 Fig04_07.cpp 6: Constant variable 'x' must be initialized in function main()
Error E2024 Fig04_07.cpp 8: Cannot modify a const object in function main()
*** 2 errors in Compile ***

**Microsoft Visual C++ compiler error messages**

Compiling...
Fig04_07.cpp
d:\fig04_07.cpp(6) : error C2734:
    'x' : const object must be initialized if not extern
d:\fig04_07.cpp(8) : error C2166:
    l-value specifies const object
Error executing cl.exe.
test.exe - 2 error(s), 0 warning(s)

**Fig. 4.7**  A `const` object must be initialized.
**Common Programming Error 4.4**  
Assigning a value to a constant variable in an executable statement is a syntax error.

Constant variables can be placed anywhere a constant expression is expected. In Fig. 4.5, constant variable `arraySize` is used to specify the size of array `s` in the declaration

```cpp
int j, s[ arraySize ];
```

**Common Programming Error 4.5**  
Only constants can be used to declare automatic and static arrays. Not using a constant for this purpose is a syntax error.

Using constant variables to specify array sizes makes programs more scalable. In Fig. 4.5, the first `for` loop could fill a 1000-element array by simply changing the value of `arraySize` in its declaration from 10 to 1000. If the constant variable `arraySize` 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.

**Software Engineering Observation 4.1**  
Defining the size of each array as a constant variable instead of a constant makes programs more scalable.

**Good Programming Practice 4.1**  
Defining the size of an array as a constant variable instead of a literal constant makes programs clearer. This technique is used to get rid of so-called magic numbers; i.e., repeatedly mentioning the size 10, for example, in array processing code for a 10-element array gives the number 10 an artificial significance and may unfortunately confuse the reader when the program includes other 10s that have nothing to do with the array size.

The program in Fig. 4.8 sums the values contained in the 12-element integer array `a`. The statement in the body of the `for` loop does the totaling. It is important to remember that the values being supplied as initializers for array `a` normally would be read into the program from the user at the keyboard. For example, the `for` structure

```cpp
for ( int j = 0; j < arraySize; j++ )
    cin >> a[ j ];
```

reads one value at a time from the keyboard and stores the value in element `a[ j ]`.

---

```cpp
1 # Fig. 4.8: fig04_08.cpp
2 // Compute the sum of the elements of the array
3 #include <iostream>
4 using std::cout;
5 using std::endl;

Fig. 4.8 Computing the sum of the elements of an array (part 1 of 2).
```
Our next example uses arrays to summarize the results of data collected in a survey. Consider the following 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 meaning awful and 10 meaning excellent). Place the 40 responses in an integer array and summarize the results of the poll.

This is a typical array application (see Fig. 4.9). We wish to summarize the number of responses of each type (i.e., 1 through 10). The array responses is a 40-element array of the students' responses. We use an 11-element array frequency to count the number of occurrences of each response. We ignore the first element, frequency[0], because it is more logical to have the response 1 increment frequency[1] than frequency[0]. This allows us to use each response directly as a subscript on the frequency array.

```
int main()
{
    const int arraySize = 12;
    int a[ arraySize ] = { 1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45 };
    int total = 0;
    for ( int i = 0; i < arraySize; i++ )
    {  
        total += a[ i ];
        cout << "Total of array element values is " << total << endl;
    }
    return 0;
}
```

**Fig. 4.8** Computing the sum of the elements of an array (part 2 of 2).

```
// Fig. 4.9: fig04_09.cpp
// Student poll program
#include <iostream>
using std::cout;
using std::endl;
#include <iomanip>
using std::setw;
int main()
{
    const int responseSize = 40, frequencySize = 11;
    int responses[ responseSize ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
    //
```

**Fig. 4.9** A student poll analysis program (part 1 of 2).
Good Programming Practice 4.2

Strive for program clarity. It is sometimes worthwhile to trade off the most efficient use of memory or processor time in favor of writing clearer programs.

Performance Tip 4.2

Sometimes performance considerations far outweigh clarity considerations.

The first for loop (lines 20 and 21) takes the responses one at a time from the array responses and increments 1 of the 10 counters (frequency[1] through frequency[10]) in the frequency array. The key statement in the loop is

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

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

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

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

```cpp
++frequency[ 2 ];
```
which increments array element two. When \texttt{answer} is 2, \texttt{responses[ answer ]} is 6, so \texttt{++frequency[ responses[ answer ] ]}; is interpreted as
\begin{verbatim}
++frequency[ 6 ];
\end{verbatim}
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 \texttt{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 either end of an array without warning. The programmer should ensure that all array references remain within the bounds of the array. C++ is an extensible language. In Chapter 8, we will extend C++ by implementing an array as a user-defined type with a class. Our new array definition will enable us to perform many operations that are not standard for C++’s built-in arrays. For example, we will be able to compare arrays directly, assign one array to another, input and output entire arrays with \texttt{cin} and \texttt{cout}, initialize arrays automatically, prevent access to out-of-range array elements and change the range of subscripts (and even their subscript type) so that the first element of an array is not required to be element 0.

\begin{itemize}
\item \textbf{Common Programming Error 4.6} Referring to an element outside the array bounds is an execution-time logic error. It is not a syntax error.
\item \textbf{Testing and Debugging Tip 4.1} When looping through an array, the array subscript should never go below 0 and should always be less than the total number of elements in the array (one less than the size of the array). Make sure that the loop-terminating condition prevents accessing elements outside this range.
\item \textbf{Testing and Debugging Tip 4.2} Programs should validate the correctness of all input values to prevent erroneous information from affecting a program’s calculations.
\item \textbf{Portability Tip 4.1} The (normally serious) effects of referencing elements outside the array bounds are system dependent. Often this results in changes to the value of an unrelated variable.
\item \textbf{Testing and Debugging Tip 4.3} When we study classes (beginning with Chapter 6), we will see how to develop a “smart array,” which automatically checks that all subscript references are in bounds at run time. Using such smart data types helps eliminate bugs.
\end{itemize}

Our next example (Fig. 4.10) reads numbers from an array and graphs the information in the form of a bar chart, or histogram—each number is printed, and then a bar consisting of that many asterisks is printed beside the number. The nested \texttt{for} loop actually draws the bars. Note the use of \texttt{endl} to end a histogram bar.

\begin{itemize}
\item \textbf{Common Programming Error 4.7} Although it is possible to use the same counter variable in a \texttt{for} loop and a second \texttt{for} loop nested inside, this is normally a logic error.
\end{itemize}
Testing and Debugging Tip 4.4

Although it is possible to modify a loop counter in a `for` body, avoid doing so, because this often leads to subtle bugs.

```cpp
// Fig. 4.10: fig04_10.cpp
// Histogram printing program
#include <iostream>
using std::cout;
using std::endl;
#include <iomanip>
using std::setw;

int main()
{
    const int arraySize = 10;
    int n[ arraySize ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };

    cout << "Element" << setw( 13 ) << "Value"
     << setw( 17 ) << "Histogram" << endl;

    for ( int i = 0; i < arraySize; i++ )
    {
        cout << setw( 7 ) << i << setw( 13 )
            << n[ i ] << setw( 9 );
        for ( int j = 0; j < n[ i ]; j++ )   // print one bar
            cout << '*';
        cout << endl;
    }
    return 0;
}
```

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
<th>Histogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19</td>
<td>***********</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>***</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>************</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>********</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>********</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>****</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td>************</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>*****</td>
</tr>
<tr>
<td>8</td>
<td>17</td>
<td>************</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>*</td>
</tr>
</tbody>
</table>

Fig. 4.10  A program that prints histograms.
In Chapter 3, we stated that we would show a more elegant method of writing the dice-rolling program of Fig. 3.8. 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. 4.11.

To this point, we have discussed only integer arrays. However, arrays may be of any type. We now discuss storing character strings in character arrays. So far, the only string-processing capability we introduced is outputting a string with `cout` and `<<`. A string such as “hello” is really an array of characters. Character arrays have several unique features.
A character array can be initialized using a string literal. For example, the declaration

```cpp
cchar 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 representation of the null character is `\0` (backslash followed by zero). All strings 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 the more tedious form

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

Because a string is 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 `cin` and `>>`. For example, the declaration

```cpp
cchar string2[ 20 ];
```

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

```cpp
  cin >> string2;
```

reads a string from the keyboard into `string2` and automatically appends the null character to the end of `string2`. Note in the preceding statement that only the name of the array is supplied; no information about the size of the array is provided. It is the programmer’s responsibility to ensure that the array into which the string is read is capable of holding any string the user types at the keyboard. `cin` reads characters from the keyboard until the first whitespace character is encountered—it does not care how large the array is. Thus, inputting data with `cin` and `>>` can insert data beyond the end of the array (see Section 5.12 for information on preventing insertion beyond the end of a `char` array).

**Common Programming Error 4.8**

Not providing `cin>>` 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 serious run-time errors.

A character array representing a null-terminated string can be output with `cout` and `<<`. The array `string2` is printed with the statement

```cpp
  cout << string2 << endl;
```

Note that `cout <<`, like `cin >>`, does not care how large the character array is. The characters of the string are printed until a terminating null character is encountered.

Figure 4.12 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 4.12 uses a for structure (lines 19 and 20) to loop through the string1 array and print the individual characters separated by spaces. The condition in the for structure, string1[i] != '\0', is true while the terminating null character has not been encountered in the string.

Chapter 3 discussed the storage class specifier static. A static local variable in a function definition exists for the duration of the program, but is only visible in the function body.

**Performance Tip 4.3**

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 improves performance.

Arrays that are declared static are initialized when the program is loaded. If a static array is not explicitly initialized by the programmer, that array is initialized to zero by the compiler when the array is created.
Figure 4.13 demonstrates function `staticArrayInit` with a local array declared
`static` and function `automaticArrayInit` with an automatic local array. Function
`staticArrayInit` is called twice. The `static` local array 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. The ele-
ments of the automatic local array 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 func-
tion is called, the array elements are reinitialized to 1, 2 and 3, because the array has auto-
matic storage class.

```cpp
// Fig. 4.13: fig04_13.cpp
// Static arrays are initialized to zero
#include <iostream>

using std::cout;
using std::endl;

void staticArrayInit( void );
void automaticArrayInit( void );

int main()
{
    cout << "First call to each function:\n";
    staticArrayInit();
    automaticArrayInit();
    cout << "\n\nSecond call to each function:\n";
    staticArrayInit();
    automaticArrayInit();
    cout << endl;
    return 0;
}

// function to demonstrate a static local array
void staticArrayInit( void )
{
    static int array1[ 3 ];
    int i;
    cout << "\nValues on entering staticArrayInit:\n";
    for ( i = 0; i < 3; i++ )
        cout << "array1[" << i << "] = " << array1[ i ] << " ";
    cout << "\nValues on exiting staticArrayInit:\n";
}
```

Fig. 4.13 Comparing `static` array initialization and automatic array initialization
(part 1 of 2).
Chapter 4  
Arrays  
257

Common Programming Error 4.9

Assuming that elements of a function’s local static array are initialized to zero every time the function is called can lead to logic errors in a program.

```cpp
for ( i = 0; i < 3; i++ )
    cout << "array1[" << i << "] = "
        << ( array1[ i ] += 5 ) << "  ";

// function to demonstrate an automatic local array
void automaticArrayInit( void )
{
    int i, array2[ 3 ] = { 1, 2, 3 };
    cout << "\n\nValues on entering automaticArrayInit:\n";
    for ( i = 0; i < 3; i++ )
        cout << "array2[" << i << "] = " << array2[ i ] << "  ";
    cout << "\nValues on exiting automaticArrayInit:\n";
    for ( i = 0; i < 3; i++ )
        cout << "array2[" << i << "] = "
            << ( array2[ i ] += 5 ) << "  ";
}
```

First call to each function:

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

Values on entering automaticArrayInit:
Values on exiting automaticArrayInit:

Second call to each function:

Values on entering staticArrayInit:
Values on exiting staticArrayInit:

Values on entering automaticArrayInit:
Values on exiting automaticArrayInit:

Fig. 4.13  Comparing static array initialization and automatic array initialization (part 2 of 2).
4.5 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

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

the function call statement

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

passes array `hourlyTemperatures` and its size to function `modifyArray`. When passing an array to a function, the array size is normally passed as well, so the function can process the specific number of elements in the array. (Otherwise, we would need to build this knowledge into the called function itself or, worse yet, place the array size in a global variable.) In Chapter 8, when we introduce the `Array` class, we will build the size of the array into the user-defined type—every `Array` object that we create will “know” its own size. Thus, when we pass an `Array` object into a function, we no longer will have to pass the size of the array as an argument.

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 value of the name of the array is 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.

Performance Tip 4.4

Passing arrays by simulated call-by-reference makes sense for performance reasons. If arrays were passed by 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 4.2

It is possible to pass an array by value (by using a simple trick we explain in Chapter 16)—this is rarely done.

Although entire arrays are passed by simulated call-by-reference, individual array elements are passed by 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 5, we show how to simulate call-by-reference for scalars (i.e., individual variables and array elements).

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

```c
void modifyArray( int b[], int arraySize )
```

indicating that `modifyArray` expects to receive the address of an array of integers in parameter `b` and the number of array elements in parameter `arraySize`. The size of the ar-
ray is not required between the array brackets. If it is included, the compiler will ignore it. Because arrays are passed by simulated call-by-reference, when the called function uses the array name \texttt{b}, it will in fact be referring to the actual array in the caller (array \texttt{hourly-Temperatures} in the preceding call). In Chapter 5, 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.

Note the strange appearance of the function prototype for \texttt{modifyArray}

\begin{verbatim}
void modifyArray( int [], int );
\end{verbatim}

This prototype could have been written

\begin{verbatim}
void modifyArray( int anyArrayName[], int anyVariableName )
\end{verbatim}

but as we learned in Chapter 3, C++ compilers ignore variable names in prototypes.

\textbf{Good Programming Practice 4.3}

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 expected to appear).

The program in Fig. 4.14 demonstrates the difference between passing an entire array and passing an array element. The program first prints the five elements of integer array \texttt{a}. Next, \texttt{a} and its size are passed to function \texttt{modifyArray}, where each of \texttt{a}'s elements is multiplied by 2. Then \texttt{a} is reprinted in \texttt{main}. As the output shows, the elements of \texttt{a} are indeed modified by \texttt{modifyArray}. Now the program prints the value of \texttt{a[3]} and passes it to function \texttt{modifyElement}. Function \texttt{modifyElement} multiplies its argument by 2 and prints the new value. Note that when \texttt{a[3]} is reprinted in \texttt{main} it has not been modified, because individual array elements are passed by call-by-value.

There may be situations in your programs in which a function should not be allowed to modify array elements. Because arrays are always passed by simulated call-by-reference, modification of values in an array is difficult to control. C++ provides the type qualifier \texttt{const} that can be used to prevent modification of array values in a function. When a function specifies an array parameter that is preceded by the \texttt{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 syntax error. This enables the programmer to correct a program so it does not attempt to modify array elements.

\begin{verbatim}
1 // Fig. 4.14: fig04_14.cpp
2 // Passing arrays and individual array elements to functions
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 #include <iomanip>
\end{verbatim}

\textbf{Fig. 4.14} Passing arrays and individual array elements to functions (part 1 of 3).
using std::setw;

void modifyArray( int [], int ); // appears strange
void modifyElement( int );

int main()
{
    const int arraySize = 5;
    int i, a[ arraySize ] = { 0, 1, 2, 3, 4 };

    cout << "Effects of passing entire array call-by-reference:"
        << "The values of the original array are:\n";
    for ( i = 0; i < arraySize; i++ )
        cout << setw( 3 ) << a[ i ];
    cout << endl;

    // array a passed call-by-reference
    modifyArray( a, arraySize );
    cout << "The values of the modified array are:"
        << "\n\n";
    for ( i = 0; i < arraySize; i++ )
        cout << setw( 3 ) << a[ i ];
    cout << "\n\n\n";
    cout << "Effects of passing array element call-by-value:"
        << "\n\nThe value of a[3] is " << a[ 3 ] << '
';
    modifyElement( a[ 3 ] );
    return 0;
}

// In function modifyArray, "b" points to the original
// array "a" in memory.
void modifyArray( int b[], int sizeOfArray )
{
    for ( int j = 0; j < sizeOfArray; j++ )
        b[ j ] *= 2;
}

// In function modifyElement, "e" is a local copy of
void modifyElement( int e )
{
    cout << "Value in modifyElement is "
        << ( e *= 2 ) << endl;
}

Fig. 4.14  Passing arrays and individual array elements to functions (part 2 of 3).
Figure 4.15 demonstrates the `const` qualifier. Function `tryToModifyArray` is defined with parameter `const int b[]`, which specifies that array `b` is constant and cannot be modified. Each of the three attempts by the function to modify array elements results in the syntax error “Cannot modify a const object.” The `const` qualifier will be discussed again in Chapter 7.

```cpp
// Fig. 4.15: fig04_15.cpp
// Demonstrating the const type qualifier
#include <iostream>
using std::cout;
using std::endl;

void tryToModifyArray( const int b[] );

int main()
{
    int a[] = { 10, 20, 30 };
    tryToModifyArray( a );
    cout << a[ 0 ] << ' ' << a[ 1 ] << ' ' << a[ 2 ] << '\n';
    return 0;
}

void tryToModifyArray( const int b[] )
{
    b[ 0 ] /= 2;    // error
    b[ 1 ] /= 2;    // error
    b[ 2 ] /= 2;    // error
}
```

**Fig. 4.15** Demonstrating the `const` type qualifier (part 1 of 2).
Common Programming Error 4.10
Forgetting that arrays are passed by reference and hence can be modified may result in a logic error.

Software Engineering Observation 4.3
The \texttt{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.

4.6 Sorting Arrays

Sorting data (i.e., placing the data into some 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 that has attracted some of the most intense research efforts in the field of computer science. In this chapter, we discuss the simplest known sorting scheme. In the exercises and in Chapter 15, we investigate more complex schemes that yield superior performance.
Performance Tip 4.5

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

The program in Fig. 4.16 sorts the values of the 10-element array `a` into ascending order. The technique we use is called the bubble sort, or the sinking sort, because the smaller values gradually “bubble” their way upward to the top of the array like air bubbles rising in water, while the larger values sink to the bottom of the array. The technique makes several passes through the array. On each pass, successive pairs of elements are compared. If a pair is in increasing order (or the values are identical), we leave the values as they are. If a pair is in decreasing order, their values are swapped in the array.

```cpp
// Fig. 4.16: fig04_16.cpp
// This program sorts an array's values into ascending order
#include <iostream>
using std::cout;
using std::endl;
#include <iomanip>
using std::setw;
int main()
{
    const int arraySize = 10;
    int a[arraySize] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
    int i, hold;
    cout << "Data items in original order\n";
    for (i = 0; i < arraySize; i++)
        cout << setw( 4 ) << a[i];
    for (int pass = 0; pass < arraySize - 1; pass++) // passes
        for (i = 0; i < arraySize - 1; i++) // one pass
            if (a[i] > a[i + 1]) // one comparison
                hold = a[i]; // one swap
                a[i] = a[i + 1];
                a[i + 1] = hold;
    cout << "\nData items in ascending order\n";
    for (i = 0; i < arraySize; i++)
        cout << setw( 4 ) << a[i];
```

Fig. 4.16   Sorting an array with bubble sort (part 1 of 2).
First the program compares $a[0]$ with $a[1]$, then $a[1]$ with $a[2]$, then $a[2]$ with $a[3]$, and so on until it completes the pass by comparing $a[8]$ to $a[9]$. Although there are 10 elements, only nine comparisons are performed. Because of the way the successive comparisons are made, a large value may move down the array many positions on a single pass, but a small value may move up only one position. On the first pass, the largest value is guaranteed to sink to the bottom element of the array, $a[9]$. On the second pass, the second largest value is guaranteed to sink to $a[8]$. On the ninth pass, the ninth largest value sinks to $a[1]$. This leaves the smallest value in $a[0]$, so only nine passes are needed to sort a 10-element array.

The sorting is performed by the nested `for` loop. If a swap is necessary, it is performed by the three assignments

```cpp
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

```cpp
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 and investigate some far more efficient sorts than the bubble sort. More advanced courses investigate sorting and searching in greater depth.

### 4.7 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. The program in Fig. 4.17 uses array `response` initialized with 99 responses (represented by constant variable `responseSize`) to a survey. Each of the responses is a number from 1 to 9. The program computes the mean, median and mode of the 99 values.
Fig. 4.17: Survey data analysis program (part 1 of 3).

// Fig. 4.17: fig04_17.cpp
// This program introduces the topic of survey data analysis.
// It computes the mean, median, and mode of the data.
#include <iostream>
#include <iomanip>
#include <assert.h>

using std::cout;
using std::endl;
using std::ios;

using std::setw;
using std::setiosflags;
using std::setprecision;

void mean( const int [], int );
void median( int [], int );
void mode( int [], int [], int );
void bubbleSort( int[], int );
void printArray( const int[], int );

int main()
{
    const int responseSize = 99;
    int frequency[10] = {0},
    response[responseSize] =
    { 6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
       7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
       7, 8, 9, 3, 9, 8, 7, 8, 7, 8,
       6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
       7, 8, 9, 9, 8, 9, 7, 8, 9, 2,
       7, 8, 9, 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 },

    mean( response, responseSize );
    median( response, responseSize );
    mode( frequency, response, responseSize );

    return 0;
}

void mean( const int answer[], int arraySize )
{
    int total = 0;
    cout << "********n Mean\n********n";
    for ( int j = 0; j < arraySize; j++ )
        total += answer[ j ];

    cout << "\nMean: " << total / arraySize;
    cout << "\n********n Mode: " << mode( response, frequency, responseSize );
    cout << "\n********n Median: " << median( response, responseSize );
}
cout << "The mean is the average value of the data\n"
<< "items. The mean is equal to the total of\n" << "all the data items divided by the number\n" "of data items (" << arraySize << ") The mean value for\nthis run is: " << total << " / " << arraySize << " = " << setiosflags( ios::fixed | ios::showpoint ) << setprecision( 4 ) << static_cast< double >( total ) / arraySize << "\n"
;
}

int median( int answer[], int size )
{
    cout << "\n********
 Median
********\n"
<< "The unsorted array of responses is"; printArray( answer, size ); bubbleSort( answer, size ); cout << "\n\nThe sorted array is"; printArray( answer, size ); cout << "\n\nThe median is element " << size / 2 << " of\nthe sorted " << size " element array.\nFor this run the median is " << answer[ size / 2 ] << "\n"
;
}

int mode( int freq[], int answer[], int size )
{
    int rating, largest = 0, modeValue = 0;
    cout << "\n********
  Mode
********\n";
    for ( rating = 1; rating <= 9; rating++ ) freq[ rating ] = 0;
    for ( int j = 0; j < size; j++ ) ++freq[ answer[ j ] ];
    for ( int j = 0; j < size; j++ ) {
        cout << "Response" << setw( 11 ) << "Frequency"
" << setw( 19 ) << "Histogram" << setw( 55 ) << "\n" << setw( 56 ) << "1  1  2  2\n" << setw( 56 ) << "5  0  5  0  5\n"
;
    for ( rating = 1; rating <= 9; rating++ ) {
        cout << setw( 8 ) << rating << freq[ rating ] << "\\n"
    for ( rating = 1; rating <= 9; rating++ ) {
        if ( freq[ rating ] > largest ) {
            largest = freq[ rating ];
            modeValue = rating;
        }
    }
Fig. 4.17   Survey data analysis program (part 2 of 3).
The mean is the arithmetic average of the 99 values. Function `mean` computes the mean by totaling the 99 elements and dividing the result by 99.

The median is the “middle value.” Function `median` determines the median by calling `bubbleSort` to sort array `response` 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 provide this capability. Function `printArray` is called to output the `response` array.

The mode is the value that occurs most frequently among the 99 responses. Function `mode` counts the number of responses of each type and then selects the value with the greatest count. This version of function `mode` does not handle a tie (see Exercise 4.14). Function `mode` also produces a histogram to aid in determining the mode graphically. Fig. 4.18 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: \( \frac{681}{99} = 6.8788 \)

Median

The unsorted array of responses is

6 7 8 9 8 7 8 9 5 9 8 7 8 7 8
6 7 8 9 3 9 8 7 8 7 8 9 8 9 8 7 9 7 8
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 4 4 4 4 4 4 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 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
9 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

<table>
<thead>
<tr>
<th>Response</th>
<th>Frequency</th>
<th>Histogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>*</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>***</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>****</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>*****</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>*********</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>*********</td>
</tr>
<tr>
<td>7</td>
<td>23</td>
<td>**********</td>
</tr>
<tr>
<td>8</td>
<td>27</td>
<td>**********</td>
</tr>
<tr>
<td>9</td>
<td>19</td>
<td>**********</td>
</tr>
</tbody>
</table>

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

Fig. 4.18  Sample run for the survey data analysis program.
4.8 Searching Arrays: Linear Search and Binary Search

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 4.33 and 4.34 at the end of this chapter ask you to implement recursive versions of the linear search and the binary search.

The linear search (Fig. 4.19) 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 must compare the search key with half the elements of the array for a value in the array. To determine that a value is not in the array, the program must compare the search key to every element in the array.

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.

```cpp
// Fig. 4.19: fig04_19.cpp
// Linear search of an array
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

int linearSearch( const int [], int, int );

int main()
{
    const int arraySize = 100;
    int a[ arraySize ], searchKey, element;
    for ( int x = 0; x < arraySize; x++ ) // create some data
        a[ x ] = 2 * x;
    cout << "Enter integer search key:" << endl;
    cin >> searchKey;
    element = linearSearch( a, searchKey, arraySize );
    if ( element != -1 )
        cout << "Found value in element " << element << endl;
    else
        cout << "Value not found" << endl;
    return 0;
}
```

Fig. 4.19 Linear search of an array (part 1 of 2).
The binary search algorithm eliminates one-half of the elements in the array being searched after each comparison. The algorithm locates the middle element of the array and compares it with the search key. If they are equal, the search key is found, and the array subscript of that element is returned. Otherwise, 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 the middle element 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 (because after each comparison, we are able to eliminate half of the array) 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 number of comparisons needed for the binary search of any sorted array can be determined by finding the first power of 2 greater than the number of elements in the array.

Performance Tip 4.6
The tremendous performance gains of the binary search over the linear search do not come without a price. Sorting an array is an expensive operation compared with searching an entire array once for one item. The overhead of sorting an array becomes worthwhile when the array will need to be searched many times at high speed.
Figure 4.20 presents the iterative version of function \texttt{binarySearch}. The function receives four arguments—an integer array \texttt{b}, an integer \texttt{searchKey}, the \texttt{low} array subscript and the \texttt{high} array subscript. If the search key does not match the middle element of a subarray, the \texttt{low} subscript or \texttt{high} subscript is adjusted so a smaller subarray can be searched. If the search key is less than the middle element, the \texttt{high} subscript is set to \texttt{middle - 1}, and the search is continued on the elements from \texttt{low} to \texttt{middle - 1}. If the search key is greater than the middle element, the \texttt{low} subscript is set to \texttt{middle + 1}, and the search is continued on the elements from \texttt{middle + 1} to \texttt{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. Function \texttt{printHeader} outputs the array subscripts and function \texttt{printRow} outputs each subarray during the binary search process. The middle element in each subarray is marked with an asterisk (*)) to indicate the element with which the search key is compared.
```cpp
return 0;
}

// Binary search
int binarySearch( const int b[], int searchKey, int low, int high,
int size )
{
int middle;

while ( low <= high ) {
    middle = ( low + high ) / 2;
    printRow( b, low, middle, high, size );
    if ( searchKey == b[ middle ] )  // match
        return middle;
    else if ( searchKey < b[ middle ] )
        high = middle - 1;        // search low end of array
    else
        low = middle + 1;         // search high end of array
}

return -1;   // searchKey not found
}

// Print a header for the output
void printHeader( int size )
{
    int i;
    cout << "\nSubscripts:\n";
    for ( i = 0; i < size; i++ )
        cout << setw( 3 ) << i << ' ';  
    cout << 'n';
    for ( i = 1; i <= 4 * size; i++ )
        cout << '-';
    cout << endl;
}

// Print one row of output showing the current
// part of the array being processed.
void printRow( const int b[], int low, int mid, int high, int size )
{
    for ( int i = 0; i < size; i++ )
        if ( i < low || i > high )
            cout << "    ";
        else if ( i == mid )           // mark middle value
            cout << setw( 3 ) << b[ i ] << '*';
Fig. 4.20  Binary search of a sorted array (part 2 of 3).
```
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. C++ compilers support at least 12 array subscripts. Fig. 4.21 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.
Every element in array \( a \) is identified in Fig. 4.21 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 4.11**

Referencing a double-subscripted array element \( a[x][y] \) incorrectly as \( a[x,y] \). Actually, \( a[x,y] \) is treated as \( a[y] \), because C++ evaluates the expression (containing a comma operator) \( x,y \) simply as \( y \) (the last of the comma-separated expressions).

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

```cpp
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

```cpp
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 4.22 demonstrates initializing double-subscripted arrays in declarations. The program declares three arrays, each with two rows and three columns. The declaration of \( \text{array1} \) 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 \( \text{array1} \) initializer list, the compiler automatically initializes the elements of the first row followed by the elements of the second row.
The declaration of `array2` provides five initializers. The initializers are assigned to the first row and 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 zero.
The declaration of `array3` 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 automatically initialized to zero. The sublist for the second row explicitly initializes the first element to 4. The last two elements are automatically initialized to zero.

The program calls function `printArray` to output each array’s elements. Notice that the function definition specifies the array parameter as `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 size of the first subscript of a multiple-subscripted array is not required either, but all subsequent subscript sizes are required. The compiler uses these sizes 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 a single-subscripted array. To locate an element in a particular row, the function 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]`, the function knows to skip the first row’s three elements in memory to get to the second row (row 1). Then, the function accesses the third element of that row (element 2).

Many common array manipulations use `for` repetition structures. For example, the following `for` structure sets all the elements in the third row of array `a` in Fig. 4.21 to zero:

```c
for ( column = 0; column < 4; column++ )
    a[ 2 ][ column ] = 0;
```

We specified the third row, and therefore we know that the first subscript is always 2. (0 is the first row subscript, and 1 is the second row subscript.) The `for` loop varies only the second subscript (i.e., the column subscript). The preceding `for` structure is equivalent to the following assignment statements:

```c
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`:

```c
total = 0;
for ( row = 0; row < 3; row++ )
    for ( column = 0; column < 4; 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 then 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.

The program of Fig. 4.23 performs several other common array manipulations on 3-by-4 array `studentGrades`. Each row of the array represents a student, and each column
represents a grade on one of the four exams the students took during the semester. The array manipulations are performed by four functions. Function `minimum` determines the lowest grade of any student for the semester. Function `maximum` determines the highest grade of any student for the semester. Function `average` determines a particular student’s semester average. Function `printArray` outputs the double-subscripted array in a neat, tabular format.

```cpp
// Fig. 4.23: fig04_23.cpp
// Double-subscripted array example
#include <iostream>
using std::cout;
using std::endl;
using std::ios;

#include <iomanip>
using std::setw;
using std::setiosflags;
using std::setprecision;

const int students = 3;   // number of students
const int exams = 4;      // number of exams

int minimum( int [[ exams ], int, int );
int maximum( int [[ exams ], int, int );
double average( int [], int );
void printArray( int [[][ exams ], int, int );

int main()
{
    int studentGrades[ students ][ exams ] =
    { { 77, 68, 86, 73 },
    { 96, 87, 89, 78 },
    { 70, 90, 86, 81 } };

    cout << "The array is:\n";
    printArray( studentGrades, students, exams );
    cout << "\n\nLowest grade: "
    << minimum( studentGrades, students, exams )
    << "\nHighest grade: "
    << maximum( studentGrades, students, exams ) << '\n';

    for ( int person = 0; person < students; person++ )
        cout << "The average grade for student " << person " is "
        << setiosflags( ios::fixed | ios::showpoint )
        << setprecision( 2 )
        << average( studentGrades[ person ], exams ) << endl;

    return 0;
}
```

Fig. 4.23  Example of using double-subscripted arrays (part 1 of 3).
// Find the minimum grade
int minimum( int grades[][ exams ], int pupils, int tests )
{
    int lowGrade = 100;
    for ( int i = 0; i < pupils; i++ )
        for ( int j = 0; j < tests; j++ )
            if ( grades[ i ][ j ] < lowGrade )
                lowGrade = grades[ i ][ j ];
    return lowGrade;
}

// Find the maximum grade
int maximum( int grades[][ exams ], int pupils, int tests )
{
    int highGrade = 0;
    for ( int i = 0; i < pupils; i++ )
        for ( int j = 0; j < tests; j++ )
            if ( grades[ i ][ j ] > highGrade )
                highGrade = grades[ i ][ j ];
    return highGrade;
}

// Determine the average grade for a particular student
double average( int setOfGrades[], int tests )
{
    int total = 0;
    for ( int i = 0; i < tests; i++ )
        total += setOfGrades[ i ];
    return static_cast< double >( total ) / tests;
}

// Print the array
void printArray( int grades[][ exams ], int pupils, int tests )
{
    cout << "                 [0]  [1]  [2]  [3]"
    for ( int i = 0; i < pupils; i++ )
        for ( int j = 0; j < tests; j++ )
            cout << setiosflags( ios::left ) << setw( 5 )
             << grades[ i ][ j ];
}

Fig. 4.23 Example of using double-subscripted arrays (part 2 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:

```c
for ( i = 0; i < pupils; i++ )
  for ( j = 0; j < tests; 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 with variable `lowGrade` in the body of the inner `for` structure. The inner `for` structure loops through the four grades of a particular row and compares each grade with `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 with variable `lowGrade`. The outer `for` structure then increments the row subscript to 2. The elements of the third row are compared with 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` takes two arguments—a single-subscripted array of test results for a particular student and the number of test results in the array. When `average` is called, the first argument is `studentGrades[ student ]`, which specifies that a particular row of the double-subscripted array `studentGrades` is to be passed to `average`. For example, the argument `studentGrades[ 1 ]` represents the four values (a single-subscripted array of grades) stored in the second row of the double-subscripted array `studentGrades`. A double-subscripted array could be considered an array with elements that are single-subscripted arrays. Function `average` calculates the sum of the array elements, divides the total by the number of test results and returns the floating-point result.
4.10 (Optional Case Study) Thinking About Objects: Identifying the Operations of a Class

In the “Thinking About Objects” sections at the ends of Chapters 2 and 3, we performed the first few steps of an object-oriented design for our elevator simulator. In Chapter 2, we identified the classes we need to implement, and we created a class diagram that models the structure of our system. In Chapter 3, we determined many of the attributes of our classes, we investigated the possible states of class Elevator and represented them in a statechart diagram and we modeled in an activity diagram the logic the elevator uses to respond to button presses.

In this section, we concentrate on determining the class operations (or behaviors) needed to implement the elevator simulator. In Chapter 5, we will concentrate on the collaborations (interactions) between objects of our classes.

An operation of a class is a service that the class provides to “clients” (users) of that class. Let us consider the operations of some real-world classes. A radio’s operations include setting its station and volume (typically invoked by a listener adjusting the radio’s controls). A car’s operations include accelerating (invoked by pressing the accelerator pedal), decelerating (invoked by pressing the brake pedal), turning and shifting gears.

Objects do not ordinarily perform their operations spontaneously. Rather, a specific operation is normally invoked when a sending object (often called a client object) sends a message to a receiving object (often called a server object) requesting that the receiving object perform that specific operation. This sounds like a member function call—precisely how messages are sent to objects in C++. In this section, we will identify many of the operations our classes need to offer to their clients in our system.

We can derive many of the operations of each class directly from the problem statement. To do so, we examine the verbs and verb phrases from the problem statement. We then relate each of these phrases to a particular class in our system (see Fig. 4.24). Many of the verb phrases in the table in Fig. 4.24 will help determine the operations of our classes.

<table>
<thead>
<tr>
<th>Class</th>
<th>Verb phrases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elevator</td>
<td>moves, arrives at a floor, resets the elevator button, sounds the elevator bell, signals its arrival to a floor, opens its door, closes its door</td>
</tr>
<tr>
<td>Clock</td>
<td>ticks every second</td>
</tr>
<tr>
<td>Scheduler</td>
<td>randomly schedules times, creates a person, tells a person to step onto a floor, verifies that a floor is unoccupied, delays creating a person by one second</td>
</tr>
<tr>
<td>Person</td>
<td>steps onto floor, presses floor button, presses elevator button, enters elevator, exits elevator</td>
</tr>
<tr>
<td>Floor</td>
<td>resets floor button, turns off light, turns on light</td>
</tr>
<tr>
<td>FloorButton</td>
<td>summons elevator</td>
</tr>
<tr>
<td>ElevatorButton</td>
<td>signals elevator to move</td>
</tr>
<tr>
<td>Door</td>
<td>(opening of door) signals person to exit elevator, (opening of door) signals person to enter elevator</td>
</tr>
</tbody>
</table>

Fig. 4.24 Verb phrases for each class in simulator.
To create operations from these verb phrases, we examine the verb phrases listed with each class. The “moves” verb listed with class `Elevator` refers to the activity in which the elevator moves between floors. Should “moves” be an operation of class `Elevator`? No message tells the elevator to move; rather, the elevator decides to move in response to a button press based on the condition that the door is closed. Therefore, “moves” does not correspond to an operation. The “arrives at a floor” phrase is also not an operation, because the elevator itself decides when to arrive on the floor, based on the time.

The “resets elevator button” phrase implies that the elevator sends a message to the elevator button telling the button to reset. Therefore, class `ElevatorButton` needs an operation to provide this service to the elevator. We place this operation in the bottom compartment of class `ElevatorButton` in our class diagram (Fig. 4.25). We represent the names of the operations as function names and include information about the return type:

```
resetButton() : void
```

The operation name is written first, followed by parentheses containing a comma-separated list of the parameters that the operation takes (in this case, none). A colon follows the parameter list, followed by the return type of the operation (in this case `void`). Note that most of our operations appear to have no parameters and to have a return type of `void`; this might change as our design and implementation processes proceed.

From the “sounds the elevator bell” phrase listed with class `Elevator`, we conclude that class `Bell` should have an operation that provides a service—ringing. We list the `ringBell` operation under class `Bell`.

When the elevator arrives at a floor, it “signals its arrival to a floor,” and the floor responds by performing its various activities (i.e., resetting the floor button and turning on the light). Therefore, class `Floor` needs an operation that provides this service. We call this operation `elevatorArrived` and place the operation name in the bottom compartment of class `Floor` in Fig. 4.25.

The remaining two verb phrases listed with class `Elevator` state that the elevator needs to open and close its door. Therefore, class `Door` needs to provide these operations. We place the `openDoor` and `closeDoor` operations in the bottom compartment of class `Door`.

Class `Clock` lists the phrase “ticks every second.” This phrase brings up an interesting point. Certainly “getting the time” is an operation that the clock provides, but is the ticking of the clock also an operation? To answer this question, we focus on our simulation will work.

---

**Fig. 4.24** Verb phrases for each class in simulator.
The problem statement indicates that the scheduler needs to know the current time to decide whether the scheduler should create a new person to step onto a floor. The elevator needs the time to decide whether it is time to arrive at a floor. We also decided that the building bears the responsibility for running the simulation and for passing the time to the scheduler and to the elevator. We now begin to see how our simulation will run. The building repeats the following steps once per second for the duration of the simulation:

1. Get the time from the clock.
2. Give the time to the scheduler so that the scheduler can create a new person, if necessary.
3. Give the time to the elevator so that the elevator can decide to arrive at a floor, if the elevator is moving.

We decided that the building has full responsibility for running all parts of the simulation. Therefore, the building must also increment the clock. The clock should be incremented once per second; then the time should be passed on to the scheduler and to the elevator.

This leads us to create two operations—\texttt{getTime} and \texttt{tick}—and list them under class \texttt{Clock}. The \texttt{getTime} operation returns as an \texttt{int} the value of the clock’s time attribute. In the preceding items 2 and 3, we see the phrases “Give the time to the scheduler” and “Give the time to the elevator.” Thus we can add the operation \texttt{processTime} to classes \texttt{Scheduler} and \texttt{Elevator}. We can also add the operation \texttt{runSimulation} to class \texttt{Building}.
Class **Scheduler** lists the verb phrases “randomly schedules times” and “delays creating a person by one second.” The scheduler decides to perform these actions itself and does not provide these services to clients. Therefore, these two phrases do not correspond to operations.

The phrase “creates person” listed with class **Scheduler** presents a special case. Although we can model an object of class **Scheduler** sending a “create” message, an object of class **Person** cannot respond to a “create” message because that object does not yet exist. The creation of objects is left to implementation details, and is not represented as an operation of a class. We discuss the creation of new objects when we discuss implementation in Chapter 7.

The phrase “tells a person to step onto a floor” listed in Fig. 4.24 means that class **Person** should have an operation that the scheduler can invoke to tell the person step onto a floor. We call this operation **stepOntoFloor** and list the operation under class **Person**.

The phrase “verifies that a floor is unoccupied” implies that class **Floor** needs to provide a service that allows objects in the system to know whether the floor is occupied or unoccupied. The operation we create for this service should return **true** if the floor is occupied and **false** if not. We place the operation

\[
isOccupied( ) : \text{bool}
\]
in the bottom compartment of class **Floor**.

Class **Person** lists the phrases “presses floor button” and “presses elevator button.” We therefore place the **pressButton** operation under classes **FloorButton** and **ElevatorButton** in our UML class diagram (Fig. 4.25). Note that we have already dealt with the fact that a person “steps onto a floor” when we analyzed the verb phrases for class **Scheduler**, so we do not need to create any operations based on the “steps onto floor” phrase listed with class **Person**. The “enters elevator” and “exits elevator” phrases listed with class **Person** suggest that class **Elevator** needs operations that correspond to these actions.1

Class **Floor** also lists “resets floor button” in its verb phrases column, so we list the appropriate **resetButton** operation under class **FloorButton**. Class **Floor** also lists “turns off light” and “turns on light,” so we create the **turnOff** and **turnOn** operations and list them under class **Light**.

The “summons elevator” phrase listed under class **FloorButton** implies that class **Elevator** needs a **summonElevator** operation. The phrase “signals elevator to move” listed with class **ElevatorButton** implies that class **Elevator** needs to provide a “move” service. Before the elevator can move, however, the elevator must close its door. Therefore, a **prepareToLeave** operation, wherein the elevator performs the necessary actions before moving, seems a more appropriate choice to list under class **Elevator**.

The phrases listed with class **Door** imply that the door sends a message to a person to tell it to exit the elevator or enter the elevator. We create two operations for class **Person** to cover these behaviors—**exitElevator** and **enterElevator**.

---

1. At this point, we can only guess what these operations do. For example, perhaps these operations model real-world elevators with a sensor that detects when passengers enter and exit. For now, we simply list these operations. We will discover what, if any, actions these operations perform when we concentrate on implementing our simulator in C++.
For now we do not overly concern ourselves with the parameters or return types; we are attempting only to gain a basic understanding of the operations of each class. As we continue our design process, the number of operations belonging to each class can vary—we might find that new operations are needed or that some current operations are unnecessary.

**Sequence Diagrams**

We can use the UML sequence diagram (see Fig. 4.26) to model our “simulation loop”—the steps from the preceding discussion that the building repeats for the duration of the simulation. The sequence diagram focuses on how messages are sent between objects over time.

Each object is represented by a rectangle at the top of the diagram. The name of the object is placed inside the rectangle. We write object names in the sequence diagram using the convention we introduced with the object diagram in the “Thinking About Objects” section at the end of Chapter 2 (Fig. 2.45). The dashed line that extends down from an object’s rectangle is that object’s lifeline. This lifeline represents the progression of time. Actions happen along an object’s lifeline in chronological order from top to bottom—an action near the top of a lifeline happens before an action near the bottom.

A message between two objects in a sequence diagram is represented as a line with a solid arrowhead that extends from the object sending that message to the object receiving that message. The message invokes the corresponding operation in the receiving object. The arrowhead points to the lifeline of the object receiving the message. The name of the message appears above the message line and should include any parameters being passed. For example, the object of class **Building** sends the **processTime** message to the object of class **Elevator**. The name of the message appears above the message line, and the name of the parameter (**currentTime**) appears inside parentheses to the right of the message; each parameter name is followed by a colon and the parameter type.

![Sequence Diagram](image-url)
If an object returns the flow of control or if an object returns a value, a return message (represented as a dashed line with an arrowhead) extends from the object returning control to the object that initially sent the message. For example, the object of class `Clock` returns `time` in response to the `getTime` message received from the object of class `Building`.

The rectangles along the objects’ lifelines—called activations—each represent the duration of an activity. An activation is initiated when an object receives a message and is denoted by a rectangle on that object’s lifeline. The height of the rectangle corresponds to the duration of the activity or activities initiated by the message—the longer the duration of the activity, the taller the rectangle.

The text to the far left of the diagram in Fig. 4.26 indicates a timing constraint. While the current time is less than the total simulation time (`currentTime < totalTime`), the objects continue sending messages to one another in the sequence modeled in the diagram.

Figure 4.27 models how the scheduler handles the time and creates new people to walk onto floors. For this diagram, we assume the scheduler has scheduled a person to walk onto each of the two floors at a time that matches the time supplied by the building. Let us follow the flow of messages through this sequence diagram.

Object `building` first sends the `processTime` message to the `scheduler`, passing the current time. The `scheduler` object must then decide whether to create a new person to step onto the first floor (represented by the `floor1` object of class `Floor`). The problem statement tells us that the scheduler must first verify that the floor is unoccupied before it can create a new person to step onto that floor. The `scheduler` object therefore sends an `isOccupied` message to the `floor1` object, to accomplish this task.

The `floor1` object returns either `true` or `false` (indicated by the dashed return message line and the `bool` type). At this point, the `scheduler` object’s lifeline splits into two parallel lifelines to represent each possible sequence of messages that the object can send, based on the value returned by the `floor1` object. An object’s lifeline can split into two or more lifelines to indicate the conditional execution of activities. A condition must be supplied for each lifeline. The new lifeline(s) run parallel to the main lifeline, and the lifelines may converge at some later point.

If the `floor1` object returns `true` (i.e., the floor is occupied), the `scheduler` calls its own `delayArrival` function, passing a parameter indicating the `floor1` arrival time needs to be rescheduled. This function is not an operation of class `Scheduler` because it is not invoked by another object. The `delayArrival` function is simply an activity class `Scheduler` performs inside an operation. Notice that when the scheduler object sends a message to itself (i.e., invokes one of its own member functions), the activation bar for that message is centered on the right edge of the current activation bar.

If the `floor1` object returns `false` (i.e., the floor is unoccupied), the `scheduler` object creates a new object of class `Person`. In a sequence diagram, when a new object is created, the new object’s rectangle is placed at a vertical position that corresponds to the time at which the object is created. An object that creates another object sends a message with the word “create” enclosed in guillemets (« »). The arrowhead of this message points to the new object’s rectangle. A large “X” at the end of an object’s lifetime denotes the destruction of that object. [Note: Our sequence diagram does not model the destruction of any objects of class `Person`; therefore, no “X” appears in the diagram. Creating and destroying objects dynamically, using C++’s `new` and `delete` operators, is discussed in Chapter 7.]
After the new object of class Person is created, the person must next step onto the first floor. Therefore, the new Person object sends a personArrives message to the floor1 object. This message notifies the floor1 object that a person is stepping onto it.

After the scheduler object has created a new object of class Person, it schedules a new arrival for floor1. The scheduler object invokes its own scheduleArrival function, and the activation bar for this call is centered on the right of the current activation bar. The scheduleArrival function is not an operation; it is an activity that class...
Scheduler performs inside an operation. At this point, the two lifelines converge. The scheduler object then handles the second floor in the same manner as the first. When the scheduler has finished with floor2, the scheduler object returns control to the building object.

In this section, we have discussed the operations of classes and introduced the UML sequence diagram to illustrate these operations. In the “Thinking About Objects” section at the end of Chapter 5, we examine how objects in a system interact with one another to accomplish a specific task, and we begin implementing our elevator simulator in C++.

**SUMMARY**

- C++ stores lists of values in arrays. An array is a group of consecutive 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. The subscript indicates the number of elements from the beginning of the array.
- A subscript may be an integer or an integer expression. Subscript expressions are 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 by declaration, by assignment and by input.
- If there are fewer initializers than elements in the array, the remaining elements are initialized 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 end with the null character (’\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.
- 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.
- Arrays are passed to functions using simulated call-by-reference—the called functions can modify the element values in the callers’ original arrays. The value of the name of the array is 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 brackets for a single-subscripted array parameter.
- C++ provides the type qualifier const that enables programs 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 is a syntax 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 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. If 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 and is acceptable for unsorted arrays.

The binary search eliminates from consideration half the elements in the array after each comparison by locating the middle element of the array and comparing it with the search key. If they are equal, the search key is found, and the array subscript of that element is returned. Otherwise, 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.

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.

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 size of the first subscript of a multiple-subscripted array is not required either, but all subsequent subscript sizes are required. The compiler uses these sizes 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 ]` named constant
- `a[ i ][ j ]` null character (`'\0'`)
- array off-by-one error
- array initializer list passing arrays to functions
- binary search of an array pass-by-reference
- bounds checking pass of a bubble sort
- bubble sort position number
- column subscript row subscript
- constant variable scalability
- const type qualifier scalar
- declare an array search an array
- double-subscripted array simulated call-by-reference
- element of an array single-subscripted array
- initialize an array sinking sort
- initializer sort an array
- linear search of an array square brackets `[`
- magic number string
- `m`-by-`n` array subscript
- multiple-subscripted array table of values
- name of an array
Chapter 4 Arrays 289

COMMON PROGRAMMING ERRORS

4.1 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. Unfortunately, this is a source of “off-by-one” errors.

4.2 Forgetting to initialize the elements of an array whose elements should be initialized is a logic error.

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

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

4.5 Only constants can be used to declare automatic and static arrays. Not using a constant for this purpose is a syntax error.

4.6 Referring to an element outside the array bounds is an execution-time logic error. It is not a syntax error.

4.7 Although it is possible to use the same counter variable in a for loop and a second for loop nested inside, this is normally a logic error.

4.8 Not providing cin >> 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 serious run-time errors.

4.9 Assuming that elements of a function’s local static array are initialized to zero every time the function is called can lead to logic errors in a program.

4.10 Forgetting that arrays are passed by reference and hence can be modified may result in a logic error.

4.11 Referencing a double-subscripted array element a[ x ][ y ] incorrectly as a[ x, y ]. Actually, a[ x, y ] is treated as a[ y ], because C++ evaluates the expression (containing a comma operator) x, y simply as y (the last of the comma-separated expressions).

GOOD PROGRAMMING PRACTICES

4.1 Defining the size of an array as a constant variable instead of a literal constant makes programs clearer. This technique is used to get rid of so-called magic numbers; i.e., repeatedly mentioning the size 10, for example, in array processing code for a 10-element array gives the number 10 an artificial significance and may unfortunately confuse the reader when the program includes other 10s that have nothing to do with the array size.
4.2 Strive for program clarity. It is sometimes worthwhile to trade off the most efficient use of memory or processor time in favor of writing clearer programs.

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

**PERFORMANCE TIPS**

4.1 If, instead of initializing an array with execution time assignment statements, you initialize the array at compile time with an array initializer list, your program will execute faster.

4.2 Sometimes performance considerations far outweigh clarity considerations.

4.3 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 improves performance.

4.4 Passing arrays by simulated call-by-reference makes sense for performance reasons. If arrays were passed by 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.

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

4.6 The tremendous performance gains of the binary search over the linear search do not come without a price. Sorting an array is an expensive operation compared with searching an entire array once for one item. The overhead of sorting an array becomes worthwhile when the array will need to be searched many times at high speed.

**PORTABILITY TIP**

4.1 The (normally serious) effects of referencing elements outside the array bounds are system dependent. Often this results in changes to the value of an unrelated variable.

**SOFTWARE ENGINEERING OBSERVATIONS**

4.1 Defining the size of each array as a constant variable instead of a constant makes programs more scalable.

4.2 It is possible to pass an array by value (by using a simple trick we explain in Chapter 6)—this is rarely done.

4.3 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.

**TESTING AND DEBUGGING TIPS**

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

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

4.3 When we study classes (beginning with Chapter 6), we will see how to develop a “smart array,” which automatically checks that all subscript references are in bounds at run time. Using such smart data types helps eliminate bugs.

4.4 Although it is possible to modify a loop counter in a `for` body, avoid doing so, because this often leads to subtle bugs.
SELF-REVIEW EXERCISES

4.1 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) The process of 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.

4.2 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 should normally be of data type float.
   c) If there are fewer initializers in an initializer list than the number of elements in the array, the remaining elements are automatically initialized 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 that function will contain the modified value when the called function completes execution.

4.3 Answer the following questions regarding an array called fractions:
   a) Define a constant variable arraySize initialized to 10.
   b) Declare an array with arraySize 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 9.
   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. Define the integer variable x as a control variable for the loop. Show the output.

4.4 Answer the following questions regarding an array called table:
   a) Declare the array to be an integer array and to have 3 rows and 3 columns. Assume that the constant variable arraySize has been defined to be 3.
   b) How many elements does the array contain?
   c) Use a for repetition structure to initialize each element of the array to the sum of its subscripts. Assume that the integer variables x and y are declared as control variables.
   d) Write a program segment to print the values of each element of an array table in tabular format with 3 rows and 3 columns. Assume that the array was initialized with the declaration

   int table[ arraySize ][ arraySize ] =
   { { 1, 2 }, { 3, 4, 5 }, { 6 } };

   and the integer variables x and y are declared as control variables. Show the output.

4.5 Find the error in each of the following program segments and correct the error:
   a) #include <iostream>;
ANSWERS TO SELF-REVIEW EXERCISES

4.1  a) Arrays.  b) Name, type.  c) Subscript.  d) Constant variable.  e) Sorting.  f) Searching.  g) Double-subscripted.

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

4.3  a) const int arraySize = 10;
    b) double fractions[ arraySize ] = { 0 };
    c) fractions[ 3 ]
    d) fractions[ 4 ]
    e) fractions[ 9 ] = 1.667;
    f) fractions[ 6 ] = 3.333;
    g) cout << setiosflags( ios::fixed | ios::showpoint )
    Output: 3.33 1.67.
    h) for ( int x = 0; x < arraySize; x++ )
       cout << "fractions[" << x << "] = " << fractions[ x ] << endl;
    Output:
    fractions[ 0 ] = 0
    fractions[ 1 ] = 0
    fractions[ 2 ] = 0
    fractions[ 3 ] = 0
    fractions[ 4 ] = 0
    fractions[ 5 ] = 0
    fractions[ 6 ] = 3.333
    fractions[ 7 ] = 0
    fractions[ 8 ] = 0
    fractions[ 9 ] = 1.667

4.4  a) int table[ arraySize ][ arraySize ];
    b) Nine.
    c) for ( x = 0; x < arraySize; x++ )
       for ( y = 0; y < arraySize; y++ )
           table[ x ][ y ] = x + y;
    d) cout << "[0][1][2]" << endl;
       for ( int x = 0; x < arraySize; x++ ) {
           cout << '[' << x << "]" << endl;
           for ( int y = 0; y < arraySize; y++ )
               cout << y;
cout << setw( 3 ) << table[ x ][ y ] << "  ";
cout << endl;
Output:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

4.5  

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

b) Error: Assigning a value to a constant variable using an assignment statement.  
Correction: Assign a value to the constant variable in a `const` declaration.

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: Array subscripting done incorrectly.  
Correction: Change the statement to `a[ 1 ][ 1 ] = 5;`

**EXERCISES**

4.6  
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 square brackets is called a _______.  
d) The names of the four elements of array `p` are _______, _______, _______ and _______.  
e) Naming an array, stating its type and specifying the number of elements in the array is called _______ the array.  
f) The process of placing the elements of an array into either ascending or descending order is called _______.  
g) 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.  
h) An `m`-by-`n` array contains _______ rows, _______ columns and _______ elements.  
i) The name of the element in row 3 and column 5 of array `d` is _______.

4.7  
State which of the following are true and which are false; for those that are false, explain why they are false.
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 at least one `for` statement.

e) A C++ program that totals the elements of a double-subscripted array must contain nested `for` statements.
4.8 Write C++ statements to accomplish each of the following:
   a) Display the value of the seventh element of character array \texttt{f}.
   b) Input a value into element 4 of single-subscripted floating-point array \texttt{b}.
   c) Initialize each of the 5 elements of single-subscripted integer array \texttt{g} to 8.
   d) Total and print the elements of floating-point array \texttt{c} of 100 elements.
   e) Copy array \texttt{a} into the first portion of array \texttt{b}. Assume \texttt{double a[11], b[34];}
   f) Determine and print the smallest and largest values contained in 99-element floating-point array \texttt{w}.

4.9 Consider a 2-by-3 integer array \texttt{t}.
   a) Write a declaration for \texttt{t}.
   b) How many rows does \texttt{t} have?
   c) How many columns does \texttt{t} have?
   d) How many elements does \texttt{t} have?
   e) Write the names of all the elements in the second row of \texttt{t}.
   f) Write the names of all the elements in the third column of \texttt{t}.
   g) Write a single statement that sets the element of \texttt{t} in row 1 and column 2 to zero.
   h) Write a series of statements that initialize each element of \texttt{t} to zero. Do not use a loop.
   i) Write a nested for structure that initializes each element of \texttt{t} to zero.
   j) Write a statement that inputs the values for the elements of \texttt{t} from the terminal.
   k) Write a series of statements that determine and print the smallest value in array \texttt{t}.
   l) Write a statement that displays the elements of the first row of \texttt{t}.
   m) Write a statement that totals the elements of the fourth column of \texttt{t}.
   n) Write a series of statements that prints the array \texttt{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.

4.10 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 $5000 in sales in a week receives $200 plus 9 percent of $5000, or a total of $650. Write a 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):
   a) $200–$299
   b) $300–$399
   c) $400–$499
   d) $500–$599
   e) $600–$699
   f) $700–$799
   g) $800–$899
   h) $900–$999
   i) $1000 and over

4.11 The bubble sort presented in Fig. 4.16 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 have 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.
4.12 Write single statements that perform the following single-subscripted array operations:
   a) Initialize the 10 elements of integer array \texttt{counts} to zero.
   b) Add 1 to each of the 15 elements of integer array \texttt{bonus}.
   c) Read 12 values for \texttt{double} array \texttt{monthlyTemperatures} from the keyboard.
   d) Print the 5 values of integer array \texttt{bestScores} in column format.

4.13 Find the error(s) in each of the following statements:
   a) Assume that: \texttt{char str[ 5 ]};
      \texttt{cin >> str; // User types hello}
   b) Assume that: \texttt{int a[ 3 ];}
   c) \texttt{double f[ 3 ] = \{ 1.1, 10.01, 100.001, 1000.0001 \};}
   d) Assume that: \texttt{double d[ 2 ][ 10 ];}
      \texttt{d[ 1, 9 ] = 2.345;}

4.14 Modify the program of Fig. 4.17 so function \texttt{mode} is capable of handling a tie for the mode value. Also modify function \texttt{median} so the two middle elements are averaged in an array with an even number of elements.

4.15 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.

4.16 Label the elements of 3-by-5 double-subscripted array \texttt{sales} to indicate the order in which they are set to zero by the following program segment:
   \begin{verbatim}
   for ( row = 0; row < 3; row++ )
       for ( column = 0; column < 5; column++ )
           sales[ row ][ column ] = 0;
   \end{verbatim}

4.17 Write a program that simulates the rolling of two dice. The program should use \texttt{rand} to roll the first die and should use \texttt{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 4.28 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).

\begin{figure}
\centering
\begin{tabular}{cccccc}
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
\end{tabular}
\caption{The 36 possible outcomes of rolling two dice.}
\end{figure}
4.18 What does the following program do?

```cpp
// ex04_18.cpp
#include <iostream>
using std::cout;
using std::endl;

int whatIsThis( int [], int );

int main()
{
    const int arraySize = 10;
    int a[ arraySize ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int result = whatIsThis( a, arraySize );
    cout << "Result is " << result << endl;
    return 0;
}

int whatIsThis( int b[], int size )
{
    if ( size == 1 )
        return b[ 0 ];
    else
        return b[ size - 1 ] + whatIsThis( b, size - 1 );
}
```

4.19 Write a program that runs 1000 games of craps and answers the following questions:
   a) How many games are won on the 1st roll, 2nd roll, ..., 20th roll, and after the 20th roll?
   b) How many games are lost on the 1st roll, 2nd roll, ..., 20th roll, and after the 20th 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?

4.20 (Airline Reservations System) A small airline has just purchased a computer for its new automated reservations system. You have been asked 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" and Please type 2 for "Economy". If the person types 1, your program should assign a seat in the first class section (seats 1-5). If the person types 2, your program should assign a seat in the economy section (seats 6-10). Your program should 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.

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 nonsmoking section (and vice versa). If yes, then make the appropriate seat assignment. If no, then print the message "Next flight leaves in 3 hours."
4.21 What does the following program do?

```cpp
// ex04_21.cpp
#include <iostream>
using std::cout;
using std::endl;

void someFunction( int [], int );

int main()
{
    const int arraySize = 10;
    int a[ arraySize ] =
        32, 27, 64, 18, 95, 14, 90, 70, 60, 37 ;
    cout << "The values in the array are:" << endl;
    someFunction( a, arraySize );
    cout << endl;
    return 0;
}

void someFunction( int b[], int size )
{
    if ( size > 0 ) {
        someFunction( &b[ 1 ], size - 1 );
        cout << b[ 0 ] << "  ";
    }
}
```

4.22 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 the following:
   a) The salesperson number
   b) The product number
   c) The total dollar value of that product sold that day

Thus, each salesperson passes in between 0 and 5 sales slips per day. Assume that the information from all of the slips for last month is available. Write a program that 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.

4.23 (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 20-by-20 array `floor` that 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:

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

Suppose that the turtle is somewhere near the center of the floor. The following “program” would draw and print a 12-by-12 square and end with the pen in the up position:

```
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.

4.24 (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. 4.29.

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? Was this close to your estimate?

b) Now let us develop a program that will move the knight around a chessboard. The board is represented by an 8-by-8 double-subscripted array `board`. Each of the squares is ini-
tialized 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. 4.25, 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:

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

vertical[ 0 ] = -1
vertical[ 1 ] = -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. 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 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 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 then 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, each center square is rated as 8, each corner square is rated as 2 and the other squares have accessibility numbers of 3, 4 or 6 as follows:

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>3</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

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 starting 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.

4.25 (Knight’s Tour: Brute-Force Approaches) In Exercise 4.24, 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 more 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 chessboard (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.

4.26 (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, i.e., no two queens are in the same row, the same column, or along the same diagonal? Use the thinking developed in Exercise 4.24 to formulate a heuristic for solving the Eight Queens problem. Run your program. (Hint: It is possible to assign a value to each square of the chessboard indicating how many squares of an empty chessboard are “eliminated” if a queen is placed in that square. Each of the corners would be assigned the value 22, as in Fig. 4.30.) 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?

4.27 (Eight Queens: Brute-Force Approaches) In this exercise, you will develop several brute-force approaches to solving the Eight Queens problem introduced in Exercise 4.26.

a) Solve the Eight Queens exercise, using the random brute-force technique developed in Exercise 4.25.

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

Fig. 4.30 The 22 squares eliminated by placing a queen in the upper-left corner.
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.

4.28 (Knight’s Tour: Closed-Tour Test) In the Knight’s Tour, a full tour occurs when the knight makes 64 moves touching each square of the 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 4.24 to test for a closed tour if a full tour has occurred.

4.29 (The Sieve of Eratosthenes) A prime integer is any integer that is evenly divisible only by itself and 1. The Sieve of Eratosthenes is a method of finding prime numbers. It operates as follows:

a) 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.

b) 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.); and so on.

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.

4.30 (Bucket Sort) A bucket sort begins with a 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 \texttt{bucketSort} that takes an integer array and the array size as arguments and performs as follows:

a) Place each value of the single-subscripted array into a row of the bucket array based on the value’s ones digit. For example, 97 is placed in row 7, 3 is placed in row 3 and 100 is placed in row 0. This is called a “distribution pass.”

b) Loop through the bucket array row by row, and copy the values back to the original array. This is called a “gathering pass.” The new order of the preceding values in the single-subscripted array is 100, 3 and 97.

c) Repeat this process for each subsequent digit position (tens, hundreds, thousands, etc.).

On the second pass, 100 is placed in row 0, 3 is placed in row 0 (because 3 has no tens digit) and 97 is placed in row 9. After the gathering pass, 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 the 3). After the last gathering pass, the original array is now in sorted order.

Note that the double-subscripted array of buckets is 10 times the size of the integer array being sorted. This sorting technique provides better performance than a bubble sort, but requires much more memory. The bubble sort requires space for only one additional element of data. This is an example of the space–time trade-off: The bucket sort uses more memory than the bubble sort, 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 swap the data between the two bucket arrays.

**RECURSION EXERCISES**

4.31 (Selection Sort) A selection sort searches an array looking for the smallest element in the array. Then, the smallest element is swapped with the first element of the array. The process is 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 performs comparably 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 sort-
ed. Write recursive function \( \text{selectionSort} \) to perform this algorithm.

4.32  (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 (if blanks are ignored) “a
man a plan a canal panama.” Write a recursive function \( \text{testPalindrome} \) that returns \( \text{true} \) if the
string stored in the array is a palindrome, and \( \text{false} \) otherwise. The function should ignore spaces
and punctuation in the string.

4.33  (Linear Search) Modify the program in Fig. 4.19 to use recursive function \( \text{linearSearch} \)
to perform a 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.

4.34  (Binary Search) Modify the program of Fig. 4.20 to use a recursive function \( \text{binary-
Search} \) 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.

4.35  (Eight Queens) Modify the Eight Queens program you created in Exercise 4.26 to solve the
problem recursively.

4.36  (Print an array) Write a recursive function \( \text{printArray} \) 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.

4.37  (Print a string backwards) Write a recursive function \( \text{stringReverse} \) that takes a char-
acter array containing a string as an argument, prints the string backwards and returns nothing. The
function should stop processing and return when the terminating null character is encountered.

4.38  (Find the minimum value in an array) Write a recursive function \( \text{recursiveMinimum} \)
that takes an integer array and the array size as arguments and returns the smallest element of the ar-
ray. The function should stop processing and return when it receives an array of 1 element.