7.2 Arrays

An array is a consecutive group of memory locations that all have 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 7.1 shows an integer array called c that contains 12 elements. You refer to any one of these elements by giving the array name followed by the particular element’s position number in square brackets ([ ]). The position number is more formally called a subscript or index (this number specifies the number of elements from the beginning of the array). The first element has subscript 0 (zero) and is sometimes called the zeroth element. Thus, the elements of array c are c[0] (pronounced “c sub zero”), c[1], c[2] and so on. The highest subscript in array c is 11, which is 1 less than the number of elements in the array (12). Array names follow the same conventions as other variable names.

![Array diagram](image)

**Fig. 7.1 | Array of 12 elements.**

A subscript must be an integer or integer expression (using any integral type). If a program uses an expression as a subscript, then the program evaluates the expression 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]. A subscripted array name is an lvalue—it can be used on the left side of an assignment, just as nonarray variable names can.

Let’s examine array c in Fig. 7.1 more closely. The name of the entire array is c. Its 12 elements are referred to as c[0] to 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’d write

\[
\text{cout} \ll c[0] + c[1] + c[2] \ll \text{endl};
\]

To divide the value of c[6] by 2 and assign the result to the variable x, we would write

\[
x = c[6] / 2;
\]
The brackets that enclose a subscript are actually an operator that has the same precedence as parentheses. Figure 7.2 shows the precedence and associativity of the operators introduced so far. The operators are shown top to bottom in decreasing order of precedence with their associativity and type.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>:: ()</td>
<td>[See parentheses caution in Fig. 2.10]</td>
<td>scope resolution</td>
</tr>
<tr>
<td>() []</td>
<td>left to right</td>
<td>function call/array access</td>
</tr>
<tr>
<td>++ -- static_cast&lt;type&gt;(operand)</td>
<td>left to right</td>
<td>unary (postfix)</td>
</tr>
<tr>
<td>++ -- + - !</td>
<td>right to left</td>
<td>unary (prefix)</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. 7.2 | Operator precedence and associativity.**

### 7.3 Declaring Arrays

Arrays occupy space in memory. To specify the type of the elements and the number of elements required by an array use a declaration of the form:

```
type arrayName[ arraySize ];
```

The compiler reserves the appropriate amount of memory. (Recall that a declaration which reserves memory is more properly known as a *definition.*) The `arraySize` must be an integer constant greater than zero. For example, to tell the compiler to reserve 12 elements for integer array `c`, use the declaration

```
type c[ 12 ]; // c is an array of 12 integers
```
Arrays can be declared to contain values of any nonreference data type. For example, an array of type `string` can be used to store character strings.

### 7.4 Examples Using Arrays

This section presents many examples that demonstrate how to declare, initialize and manipulate arrays.

#### 7.4.1 Declaring an Array and Using a Loop to Initialize the Array’s Elements

The program in Fig. 7.3 declares 10-element integer array `n` (line 9). Lines 12–13 use a `for` statement to initialize the array elements to zeros. Like other automatic variables, automatic arrays are *not* implicitly initialized to zero although static arrays are. The first output statement (line 15) displays the column headings for the columns printed in the subsequent `for` statement (lines 18–19), which prints the array in tabular format. Remember that `setw` specifies the field width in which only the next value is to be output.

```cpp
int main()
{
    int n[10]; // n is an array of 10 integers
    for (int i = 0; i < 10; ++i)
        n[i] = 0; // set element at location i to 0
    cout << "Element" << setw(13) << "Value" << endl;
    // output each array element's value
    for (int j = 0; j < 10; ++j)
        cout << setw(7) << j << setw(13) << n[j] << endl;
}
```

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>

**Fig. 7.3** | Initializing an array’s elements to zeros and printing the array.
7.4.2 Initializing an Array in a Declaration with an Initializer List

The elements of an array also can be initialized in the array declaration by following the array name with an equals sign and a brace-delimited comma-separated list of initializers. The program in Fig. 7.4 uses an initializer list to initialize an integer array with 10 values (line 10) and prints the array in tabular format (lines 12–16).

```cpp
// Fig. 7.4: fig07_04.cpp
// Initializing an array in a declaration.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    // use initializer list to initialize array n
    int n[10] = { 32, 27, 64, 18, 95, 14, 90, 60, 37 };

    cout << "Element" << setw(13) << "Value" << endl;
    for (int i = 0; i < 10; ++i)
        cout << setw(7) << i << setw(13) << n[i] << endl;
    // end main
}
```

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>27</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>95</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>90</td>
</tr>
<tr>
<td>7</td>
<td>70</td>
</tr>
<tr>
<td>8</td>
<td>60</td>
</tr>
<tr>
<td>9</td>
<td>37</td>
</tr>
</tbody>
</table>

Fig. 7.4 | Initializing an array in a declaration.

If there are fewer initializers than array elements, the remaining array elements are initialized to zero. For example, the elements of array `n` in Fig. 7.3 could have been initialized to zero with the declaration

```cpp
int n[10] = {}; // initialize elements of array n to 0
```

which initializes the elements to zero, because there are fewer initializers (none in this case) than array elements. This technique can be used only in the array’s declaration, whereas the initialization technique shown in Fig. 7.3 can be used repeatedly during program execution to “reinitialize” an array’s elements.

If the array size is omitted from a declaration with an initializer list, the compiler sizes the array to the number of elements in the initializer list. For example,

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

creates a five-element array.
If the array size and an initializer list are specified in an array declaration, the number of initializers must be less than or equal to the array size. The array declaration

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

causes a compilation error, because there are six initializers and only five array elements.

### 7.4.3 Specifying an Array’s Size with a Constant Variable and Setting Array Elements with Calculations

Figure 7.5 sets the elements of a 10-element array \( s \) to the even integers 2, 4, 6, …, 20 (lines 14–15) and prints the array in tabular format (lines 17–21). These numbers are generated (line 15) by multiplying each successive value of the loop counter by 2 and adding 2.

```cpp
// Fig. 7.5: fig07_05.cpp
// Set array s to the even integers from 2 to 20.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    // constant variable can be used to specify array size
    const int arraySize = 10;

    int s[ arraySize ]; // array s has 10 elements

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

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

    // output contents of array s in tabular format
    for ( int j = 0; j < arraySize; ++j )
        cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << endl;

    // end main
}
```

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>20</td>
</tr>
</tbody>
</table>

**Fig. 7.5** Generating values to be placed into elements of an array.

Line 10 uses the `const` qualifier to declare a so-called `constant variable` `arraySize` with the value 10. Constant variables `must` be initialized with a constant expression when
they’re declared and cannot be modified thereafter (as shown in Fig. 7.6 and Fig. 7.7). Constant variables are also called named constants or read-only variables.

### Common Programming Error 7.2
Not initializing a constant variable when it’s declared is a compilation error.

### Common Programming Error 7.3
Assigning a value to a constant variable in an executable statement is a compilation error.

```cpp
// Fig. 7.6: fig07_06.cpp
// Using a properly initialized constant variable.
#include <iostream>
using namespace std;

int main()
{
    const int x = 7; // initialized constant variable
    cout << "The value of constant variable x is: " << x << endl;
} // end main
```

**Fig. 7.6** Using a properly initialized constant variable.

```cpp
// Fig. 7.7: fig07_07.cpp
// A const variable must be initialized.

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

**Microsoft Visual C++ compiler error message:**

```
C:\cpphtp8\examples\ch07\fig07_07.cpp(6) : error C2734: 'x' : const object must be initialized if not extern
C:\cpphtp8\examples\ch07\fig07_07.cpp(8) : error C3892: 'x' : you cannot assign to a variable that is const
```

**GNU C++ compiler error message:**

```
fig07_07.cpp:6: error: uninitialized const 'x'
fig07_07.cpp:8: error: assignment of read-only variable 'x'
```

**Fig. 7.7** A const variable must be initialized.
In Fig. 7.7, the compilation error produced by Microsoft Visual C++ refers to the `int` variable `x` as a “const object.” The ISO/IEC C++ standard defines an “object” as any “region of storage.” Like objects of classes, fundamental-type variables also occupy space in memory, so they’re often referred to as “objects.”

Constant variables can be placed anywhere a constant expression is expected. In Fig. 7.5, constant variable `arraySize` specifies the size of array `s` in line 12.

### Common Programming Error 7.4

Only constants can be used to declare the size of automatic and static arrays. Not using a constant for this purpose is a compilation error.

Using constant variables to specify array sizes makes programs more scalable. In Fig. 7.5, the first `for` statement 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 lines 12, 14 and 20 of the program to scale the program to handle 1000 array elements. As programs get larger, this technique becomes more useful for writing clearer, easier-to-modify programs.

### Software Engineering Observation 7.1

Defining the size of each array as a constant variable instead of a literal constant can make programs more scalable.

### Good Programming Practice 7.1

Defining the size of an array as a constant variable instead of a literal constant makes programs clearer. This technique eliminates so-called magic numbers. For example, repeatedly mentioning the size 10 in array-processing code for a 10-element array gives the number 10 an artificial significance and can be confusing when the program includes other 10s that have nothing to do with the array size.

### 7.4.4 Summing the Elements of an Array

Often, the elements of an array represent a series of values to be used in a calculation. For example, if the elements of an array represent exam grades, a professor may wish to total the elements of the array and use that sum to calculate the class average for the exam.

The program in Fig. 7.8 sums the values contained in the 10-element integer array `a`. The program declares, creates and initializes the array in line 9. The `for` statement (lines 13–14) performs the calculations. The values being supplied as initializers for array `a` also could be read into the program from the user at the keyboard, or from a file on disk (see Chapter 17, File Processing). For example, the `for` statement

```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].`

---

1 // Fig. 7.8: fig07_08.cpp
2 // Compute the sum of the elements of the array.
3 #include <iostream>

**Fig. 7.8** | Computing the sum of the elements of an array. (Part 1 of 2.)
Chapter 7  Arrays and Vectors

7.4.5 Using Bar Charts to Display Array Data Graphically

Many programs present data to users in a graphical manner. For example, numeric values are often displayed as bars in a bar chart. In such a chart, longer bars represent proportionally larger numeric values. One simple way to display numeric data graphically is with a bar chart that shows each numeric value as a bar of asterisks (*).

Professors often like to examine the distribution of grades on an exam. A professor might graph the number of grades in each of several categories to visualize the grade distribution. Suppose the grades were 87, 68, 94, 100, 83, 78, 85, 91, 76 and 87. There was one grade of 100, two grades in the 90s, four grades in the 80s, two grades in the 70s, one grade in the 60s and no grades below 60. Our next program (Fig. 7.9) stores this grade distribution data in an array of 11 elements, each corresponding to a category of grades. For example, n[0] indicates the number of grades in the range 0–9, n[7] indicates the number of grades in the range 70–79 and n[10] indicates the number of grades of 100. The two versions of class GradeBook later in the chapter (Figs. 7.15–7.16 and Figs. 7.22–7.23) contain code that calculates these grade frequencies based on a set of grades. For now, we manually create the array by looking at the set of grades.

```
4  using namespace std;
5  int main()
6  {
7      const int arraySize = 10; // constant variable indicating size of array
8      int a[ arraySize ] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
9      int total = 0;
10     // sum contents of array a
11     for ( int i = 0; i < arraySize; ++i )
12        total += a[ i ];
13     cout << "Total of array elements: " << total << endl;
14 } // end main
```

Fig. 7.8 | Computing the sum of the elements of an array. (Part 2 of 2.)

```
// Fig. 7.9: fig07_09.cpp
// Bar chart printing program.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    const int arraySize = 11;
    int n[ arraySize ] = { 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };

    Total of array elements: 849
```

Fig. 7.9 | Bar chart printing program. (Part 1 of 2.)
7.4 Examples Using Arrays

The program reads the numbers from the array and graphs the information as a bar chart, displaying each grade range followed by a bar of asterisks indicating the number of grades in that range. To label each bar, lines 18–23 output a grade range (e.g., "70–79: ") based on the current value of counter variable \( i \). The nested for statement (lines 26–27) outputs the bars. Note the loop-continuation condition in line 26 (\( \text{stars} < \text{n}[\ i] \)). Each time the program reaches the inner for, the loop counts from 0 up to \( \text{n}[\ i] \), thus using a value in array \( \text{n} \) to determine the number of asterisks to display. In this example, \( \text{n}[0]–\text{n}[5] \) contain zeros because no students received a grade below 60. Thus, the program displays no asterisks next to the first six grade ranges.

### 7.4.6 Using the Elements of an Array as Counters

Sometimes, programs use counter variables to summarize data, such as the results of a survey. In Fig. 6.9, we used separate counters in our die-rolling program to track the number of occurrences of each side of a die as the program rolled the die 6,000,000 times. An array version of this program is shown in Fig. 7.10.
Chapter 7  Arrays and Vectors

Figure 7.10 uses the array frequency (line 12) to count the occurrences of each side of the die. The single statement in line 18 of this program replaces the switch statement in lines 25–47 of Fig. 6.9. Line 18 uses a random value to determine which frequency element to increment during each iteration of the loop. The calculation in line 18 produces a random subscript from 1 to 6, so array frequency must be large enough to store six counters. However, we use a seven-element array in which we ignore frequency[0]—it’s more logical to have the die face value 1 increment frequency[1] than frequency[0]. Thus, each face value is used as a subscript for array frequency. We also replace lines 51–56 of Fig. 6.9 by looping through array frequency to output the results (lines 23–25).

7.4.7 Using Arrays to Summarize Survey Results

Our next example uses arrays to summarize the results of data collected in a survey. Consider the following problem statement:
Twenty students were asked to rate on a scale of 1 to 5 the quality of the food in the student cafeteria, with 1 being "awful" and 5 being "excellent." Place the 20 responses in an integer array and determine the frequency of each rating.

This is a typical array-processing application (Fig. 7.11). We wish to summarize the number of responses of each type (that is, 1–5). The array responses (lines 14–15) is a 20-element integer array of the students' responses to the survey. The array responses is declared const, as its values do not (and should not) change. We use a six-element array frequency (line 18) to count the number of occurrences of each response. Each element of the array is used as a counter for one of the survey responses and is initialized to zero. As in Fig. 7.10, we ignore frequency[0].

The first for statement (lines 22–23) takes the responses one at a time from the array responses and increments one of the five counters in the frequency array (frequency[1]...frequency[5]).
to \texttt{frequency[5]}. The key statement in the loop is line 23, which increments the appropriate frequency counter, depending on the value of \texttt{responses[answer]}.

Let’s consider several iterations of the \texttt{for} loop. When control variable \texttt{answer} is 0, the value of \texttt{responses[answer]} is the value of \texttt{responses[0]} (i.e., 1 in line 14), so the program interprets \texttt{++frequency[responses[answer]]} as

\begin{verbatim}
++frequency[ 1 ]
\end{verbatim}

which increments the value in array element 1. To evaluate the expression, start with the value in the innermost set of square brackets (\texttt{answer}). Once you know \texttt{answer}’s value (which is the value of the loop control variable in line 23), plug it into the expression and evaluate the next outer set of square brackets (i.e., \texttt{responses[answer]}, which is a value selected from the responses array in lines 14–17). Then use the resulting value as the subscript for the \texttt{frequency} array to specify which counter to increment.

When \texttt{answer} is 1, \texttt{responses[answer]} is the value of \texttt{responses[1]}, which is 2, so the program interprets \texttt{++frequency[responses[answer]]} as

\begin{verbatim}
++frequency[ 2 ]
\end{verbatim}

which increments array element 2.

When \texttt{answer} is 2, \texttt{responses[answer]} is the value of \texttt{responses[2]}, which is 5, so the program interprets \texttt{++frequency[responses[answer]]} as

\begin{verbatim}
++frequency[ 5 ]
\end{verbatim}

which increments array element 5, and so on. Regardless of the number of responses processed in the survey, the program requires only an six-element array (ignoring element zero) to summarize the results, because all the response values are between 1 and 5 and the subscript values for an six-element array are 0 through 5.

If the data in \texttt{responses} contained an invalid value, such as 13, the program would have attempted to add 1 to \texttt{frequency[13]}, which is outside the bounds of the array. \texttt{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. You should ensure that all array references remain within the bounds of the array.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Common Programming Error 7.5} & \textbf{Error-Prevention Tip 7.1} \\
\hline
Referring to an element outside the array bounds is an execution-time logic error. It isn’t a syntax error. & When looping through an array, the index should never go below 0 and should always be less than the total number of array elements (one less than the size of the array). Make sure that the loop-termination condition prevents accessing elements outside this range. \\
\hline
\end{tabular}
\end{table}

\texttt{C++} is an extensible language. Section 7.11 presents \texttt{C++ Standard Library class template vector}, which enables you to perform many operations that are not available for built-in arrays. For example, we’ll be able to compare vectors directly and assign one vector to another. In Chapter 11, we extend \texttt{C++} further by implementing an array as a class of our own. This new array definition will enable us to input and output entire arrays with \texttt{cin} and \texttt{cout}, initialize arrays when they’re created and prevent access to out-of-range array elements. We’ll even be able to use noninteger subscripts.
7.4 Examples Using Arrays

7.4.8 Static Local Arrays and Automatic Local Arrays

Chapter 6 discussed the storage-class specifier static. A static local variable in a function definition exists for the program’s duration but is visible only in the function’s body.

**Error-Prevention Tip 7.2**

In Chapter 11, we’ll see how to develop a class representing a “smart array,” which checks that all subscript references are in bounds at runtime. Using such smart data types helps eliminate bugs.

**Performance Tip 7.1**

We can apply static to a local array declaration so that it not created and initialized each time the program calls the function and is not destroyed each time the function terminates. This can improve performance, especially when using large arrays.

A program initializes static local arrays when their declarations are first encountered. If a static array is not initialized explicitly by you, each element of that array is initialized to zero by the compiler when the array is created. Recall that C++ does not perform such default initialization for automatic variables.

Figure 7.12 demonstrates function staticArrayInit (lines 23–39) with a static local array (line 26) and function automaticArrayInit (lines 42–58) with an automatic local array (line 45).

```cpp
// Fig. 7.12: fig07_12.cpp
// Static arrays are initialized to zero.
#include <iostream>
using namespace std;

const int arraySize = 3;

int main()
{
    cout << "First call to each function:\n";
    staticArrayInit();
    automaticArrayInit();

    cout << "\n\nSecond call to each function:\n";
    staticArrayInit();
    automaticArrayInit();
    cout << endl;
} // end main

// function to demonstrate a static local array
void staticArrayInit( void )
{
    // initializes elements to 0 first time function is called
    static int array1[ arraySize ]; // static local array
```

Fig. 7.12 | static array initialization and automatic array initialization. (Part 1 of 2.)
cout << "\nValues on entering staticArrayInit:\n";

// output contents of array1
for ( int i = 0; i < arraySize; ++i )
    cout << "array1[" << i << "] = " << array1[i] << " \n";

cout << "\nValues on exiting staticArrayInit:\n";

// modify and output contents of array1
for ( int j = 0; j < arraySize; ++j )
    cout << "array1[" << j << "] = " << ( array1[j] += 5 ) << " \n";
}
// end function staticArrayInit

// function to demonstrate an automatic local array
void automaticArrayInit( void )
{
    // initializes elements each time function is called
    int array2[ arraySize ] = { 1, 2, 3 }; // automatic local array
    cout << "\nValues on entering automaticArrayInit:\n";

    // output contents of array2
    for ( int i = 0; i < arraySize; ++i )
        cout << "array2[" << i << "] = " << array2[i] << " \n";

    cout << "Values on exiting automaticArrayInit:\n";

    // modify and output contents of array2
    for ( int j = 0; j < arraySize; ++j )
        cout << "array2[" << j << "] = " << ( array2[j] += 5 ) << " \n";
}
// end function automaticArrayInit

First call to each function:
Values on entering staticArrayInit:
array1[0] = 0 array1[1] = 0 array1[2] = 0
Values on exiting staticArrayInit:

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:
Function `staticArrayInit` is called twice (lines 13 and 17). The static local array is initialized to zero by the compiler the first time the function is called. 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 modified values stored during the first function call. Function `automaticArrayInit` also is called twice (lines 14 and 18). The elements of the automatic local array are initialized (line 45) with the values 1, 2 and 3. The function prints the array, adds 5 to each element and prints the array again. The second time the function is called, the array elements are reinitialized to 1, 2 and 3. The array has automatic storage class, so the array is recreated and reinitialized during each call to `automaticArrayInit`.

### Common Programming Error 7.6

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

### 7.5 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

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

the function call

```cpp
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 Section 7.11, when we present C++ Standard Library class template `vector` to represent a more robust type of array, you’ll see that the size of a vector is built in—every vector object “knows” its own size, which can be obtained by invoking the vector object’s `size` member function. Thus, when we pass a `vector object` into a function, we won’t have to pass the size of the vector as an argument.

C++ passes arrays to functions 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 in the computer’s memory 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 in memory. Therefore, when the called function modifies array elements in its function body, it’s modifying the actual elements of the array in their original memory locations.

### Performance Tip 7.2

Passing arrays by reference makes sense for performance reasons. Passing by value would require copying each element. For large, frequently passed arrays, this would be time consuming and would require considerable storage for the copies of the array elements.

### Software Engineering Observation 7.2

It’s possible to pass an array by value by simply embedding it as a data member of a class and passing an object of the class, which defaults to pass-by-value.
Although entire arrays are passed by reference, individual array elements are passed by value exactly as simple variables are. 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 6, we showed how to pass individual variables and array elements by reference with references—in Chapter 8, we show how to pass them by reference with pointers.

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

```cpp
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 array’s size is not required in the array brackets. If it’s included, the compiler ignores it; thus, arrays of any size can be passed to the function. C++ passes arrays to functions by reference—when the called function uses the array name b, it refers to the actual array in the caller (i.e., array hourlyTemperatures discussed at the beginning of this section).

Note the strange appearance of the function prototype for modifyArray

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

This prototype could have been written (for documentation purposes)

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

but, as we learned in Chapter 3, C++ compilers ignore variable names in prototypes. Remember, the prototype tells the compiler the number of arguments and the type of each argument (in the order in which the arguments are expected to appear).

The program in Fig. 7.13 demonstrates the difference between passing an entire array and passing an array element. Lines 19–20 print the five original elements of integer array a. Line 25 passes a and its size to function modifyArray (lines 40–45), which multiplies each of a’s elements by 2 (through parameter b). Then, lines 29–30 print array a again in main. As the output shows, the elements of a are indeed modified by modifyArray. Next, line 33 prints the value of a[3], then line 35 passes element a[3] to function modifyElement (lines 49–53), which multiplies its parameter by 2 and prints the new value. When line 36 prints a[3] again in main, the value has not been modified, because individual array elements are passed by value.

```cpp
// Fig. 7.13: fig07_13.cpp
// Passing arrays and individual array elements to functions.
#include <iostream>
#include <iomanip>
using namespace std;

void modifyArray( int [], int ); // appears strange; array and size
void modifyElement( int ); // receive array element value

int main()
{
```

Fig. 7.13 | Passing arrays and individual array elements to functions. (Part 1 of 2.)
7.5 Passing Arrays to Functions

const int arraySize = 5; // size of array a
int a[ arraySize ] = { 0, 1, 2, 3, 4 }; // initialize array a

cout << "Effects of passing entire array by reference:
" << "\nThe values of the original array are:\n";

// output original array elements
for ( int i = 0; i < arraySize; ++i )
    cout << setw( 3 ) << a[ i ];
cout << endl;

// pass array a to modifyArray by reference
modifyArray( a, arraySize );

cout << "The values of the modified array are:\n";

// output modified array elements
for ( int j = 0; j < arraySize; ++j )
    cout << setw( 3 ) << a[ j ];
cout << endl;

cout << "\n\nEffects of passing array element by value:


} // end main

void modifyArray( int b[], int sizeOfArray )
{
    // multiply each array element by 2
    for ( int k = 0; k < sizeOfArray; ++k )
        b[ k ] *= 2;
} // end function modifyArray

void modifyElement( int e )
{
    // multiply parameter by 2
    cout << "Value of element in modifyElement: " << ( e *= 2 ) << endl;
} // end function modifyElement

Effects of passing entire array by reference:
The values of the original array are:
0 1 2 3 4
The values of the modified array are:
0 2 4 6 8

Effects of passing array element by value:
Value of element in modifyElement: 12

Fig. 7.13 | Passing arrays and individual array elements to functions. (Part 2 of 2.)
There may be situations in your programs in which a function should *not* be allowed to modify array elements. The type qualifier `const` can be used to prevent modification of array values in the caller by code in a called function. When a function specifies an array parameter that’s preceded by the `const` qualifier, the elements of the array become constant in the function body, and any attempt to modify an element of the array in the function body results in a compilation error. This enables you to prevent accidental modification of array elements in the function’s body.

Figure 7.14 demonstrates the `const` qualifier applied to an array parameter. Function `tryToModifyArray` (lines 18–21) is defined with parameter `const int b[]`, which specifies that array `b` is constant and cannot be modified. The attempt by the function to modify array `b`’s element 0 (line 20) results in a compilation error. Some compilers, for example, produce an error like “Cannot modify a const object.” This message indicates that using a `const` object (`b[0]`) as an *lvalue* is an error—you cannot assign a new value to a `const` object.

Software Engineering Observation 7.3

Applying the `const` type qualifier to an array parameter in a function definition to prevent the original array from being modified in the function body is another example of the principle of least privilege. Functions should not be given the capability to modify an array unless it’s absolutely necessary.

```cpp
#include <iostream>
using namespace std;

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

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

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

Microsoft Visual C++ compiler error message:

c:\cpphtp8\examples\ch07\fig07_14\fig07_14.cpp(20): error C3892: 'b': you cannot assign to a variable that is const

GNU C++ compiler error message:

fig07_14.cpp:20: error: assignment of read-only location