CHAPTER 7

Arrays

TOPICS

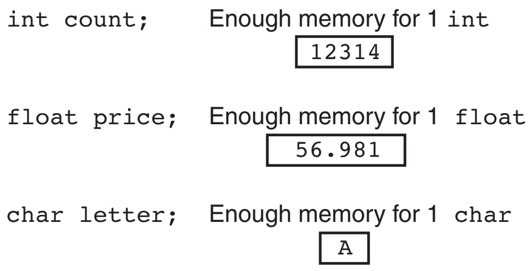
7.1 Arrays Hold Multiple Values
7.2 Accessing Array Elements
7.3 No Bounds Checking in C++
7.4 Array Initialization
7.5 The Range-Based `for` Loop
7.6 Processing Array Contents
7.7 Focus on Software Engineering: Using Parallel Arrays
7.8 Arrays as Function Arguments
7.9 Two-Dimensional Arrays
7.10 Arrays with Three or More Dimensions
7.11 Focus on Problem Solving and Program Design: A Case Study
7.12 If You Plan to Continue in Computer Science: Introduction to the STL `vector`

7.1 Arrays Hold Multiple Values

CONCEPT: An array allows you to store and work with multiple values of the same data type.

The variables you have worked with so far are designed to hold only one value at a time. Each of the variable definitions in Figure 7-1 causes only enough memory to be reserved to hold one value of the specified data type.

Figure 7-1



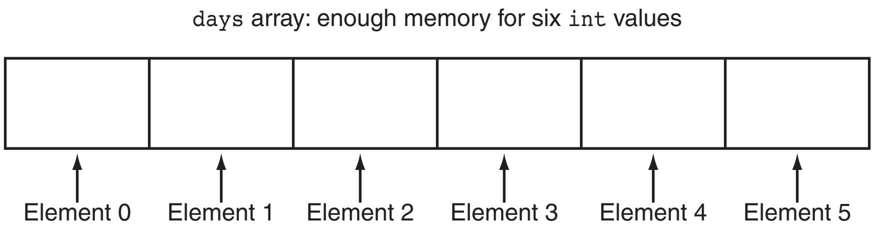
```
int count;
float price;
char letter;
```

An array works like a variable that can store a group of values, all of the same type. The values are stored together in consecutive memory locations. Here is a definition of an array of integers:

```
int days[6];
```

The name of this array is `days`. The number inside the brackets is the array's *size declarator*. It indicates the number of *elements*, or values, the array can hold. The `days` array can store six elements, each one an integer. This is depicted in Figure 7-2.

Figure 7-2



An array's size declarator must be a constant integer expression with a value greater than zero. It can be either a literal, as in the previous example, or a named constant, as shown in the following:

```
const int NUM_DAYS = 6;
int days[NUM_DAYS];
```

Arrays of any data type can be defined. The following are all valid array definitions:

```
float temperatures[100];     // Array of 100 floats
string names[10];            // Array of 10 string objects
long units[50];              // Array of 50 long integers
double sizes[1200];          // Array of 1200 doubles
```

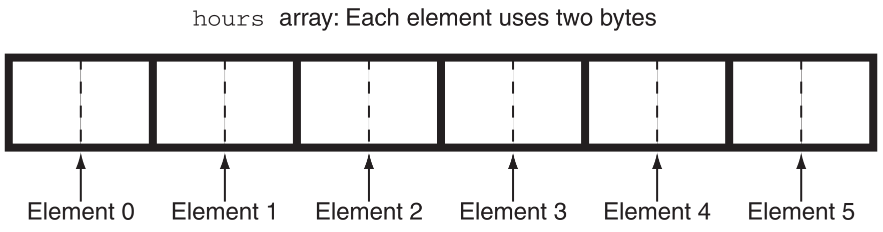
Memory Requirements of Arrays

The amount of memory used by an array depends on the array's data type and the number of elements. The `hours` array, defined here, is an array of six `shorts`.

```
short hours[6];
```

On a typical PC, a `short` uses two bytes of memory, so the `hours` array would occupy 12 bytes. This is shown in Figure 7-3.

Figure 7-3



The size of an array can be calculated by multiplying the size of an individual element by the number of elements in the array. Table 7-1 shows the typical sizes of various arrays.

Table 7-1

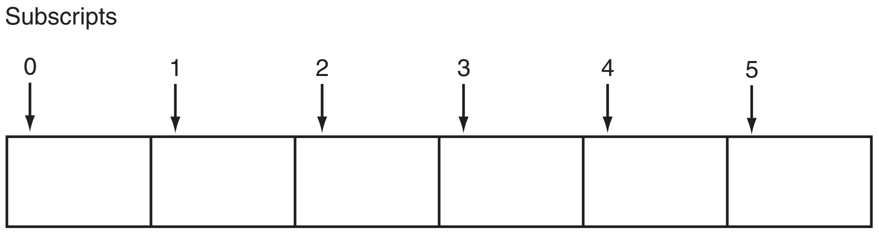
Array Definition	Number of Elements	Size of Each Element	Size of the Array
`char letters[25];`	25	1 byte	25 bytes
`short rings[100];`	100	2 bytes	200 bytes
`int miles[84];`	84	4 bytes	336 bytes
`float temp[12];`	12	4 bytes	48 bytes
`double distance[1000];`	1000	8 bytes	8000 bytes

7.2 Accessing Array Elements

CONCEPT: The individual elements of an array are assigned unique subscripts. These subscripts are used to access the elements.

Even though an entire array has only one name, the elements may be accessed and used as individual variables. This is possible because each element is assigned a number known as a *subscript*. A subscript is used as an index to pinpoint a specific element within an array. The first element is assigned the subscript 0, the second element is assigned 1, and so forth. The six elements in the array `hours` would have the subscripts 0 through 5. This is shown in Figure 7-4.

Figure 7-4



NOTE: Subscript numbering in C++ always starts at zero. The subscript of the last element in an array is one less than the total number of elements in the array. This means that in the array shown in Figure 7-4, the element `hours[6]` does not exist. `hours[5]` is the last element in the array.

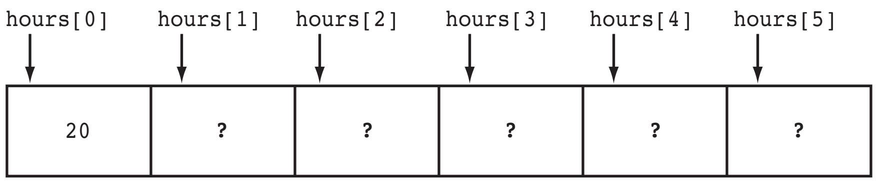
Each element in the `hours` array, when accessed by its subscript, can be used as a `short` variable. Here is an example of a statement that stores the number 20 in the first element of the array:

```
hours[0] = 20;
```

NOTE: The expression `hours[0]` is pronounced "hours sub zero." You would read this assignment statement as "hours sub zero is assigned twenty."

Figure 7-5 shows the contents of the array `hours` after the statement assigns 20 to `hours[0]`.

Figure 7-5



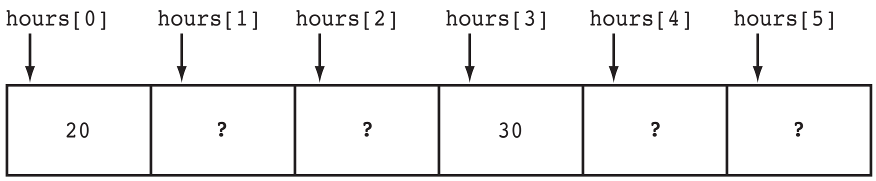
NOTE: Because values have not been assigned to the other elements of the array, question marks will be used to indicate that the contents of those elements are unknown. If an array is defined globally, all of its elements are initialized to zero by default. Local arrays, however, have no default initialization value.

The following statement stores the integer 30 in `hours[3]`.

```
hours[3] = 30;
```

Figure 7-6 shows the contents of the array after the previous statement executes:

Figure 7-6



NOTE: Understand the difference between the array size declarator and a subscript. The number inside the brackets of an array definition is the size declarator. The number inside the brackets of an assignment statement or any statement that works with the contents of an array is a subscript.

Inputting and Outputting Array Contents

Array elements may be used with the `cin` and `cout` objects like any other variable. Program 7-1 shows the array `hours` being used to store and display values entered by the user.

Program 7-1

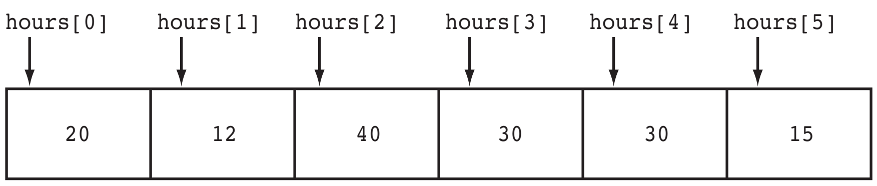
```
// This program asks for the number of hours worked
// by six employees. It stores the values in an array.
#include <iostream>
using namespace std;

int main()
{
    const int NUM_EMPLOYEES = 6;
    int hours[NUM_EMPLOYEES];

    // Get the hours worked by each employee.
    cout << "Enter the hours worked by "
         << NUM_EMPLOYEES << " employees: ";
    cin >> hours[0];
    cin >> hours[1];
    cin >> hours[2];
    cin >> hours[3];
    cin >> hours[4];
    cin >> hours[5];

    // Display the values in the array.
    cout << "The hours you entered are:";
    cout << " " << hours[0];
    cout << " " << hours[1];
    cout << " " << hours[2];
    cout << " " << hours[3];
    cout << " " << hours[4];
    cout << " " << hours[5] << endl;
    return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter the hours worked by 6 employees: 20 12 40 30 30 15 [Enter]
The hours you entered are: 20 12 40 30 30 15
```

Figure 7-7 shows the contents of the array hours with the values entered by the user in the example output above.

Figure 7-7



Even though the size declarator of an array definition must be a constant or a literal, subscript numbers can be stored in variables. This makes it possible to use a loop to "cycle through" an entire array, performing the same operation on each element. For example, look at the following code:

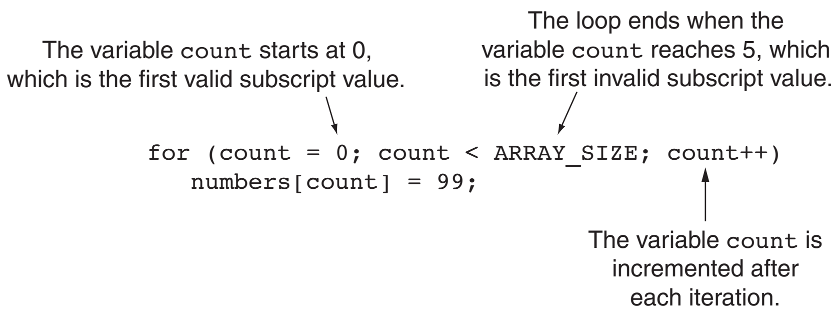
```
const int ARRAY_SIZE = 5;
int numbers[ARRAY_SIZE];

for (int count = 0; count < ARRAY_SIZE; count++)
   numbers[count] = 99;
```

VideoNote
Accessing Array Elements With a Loop

This code first defines a constant `int` named `ARRAY_SIZE` and initializes it with the value 5. Then it defines an `int` array named `numbers`, using `ARRAY_SIZE` as the size declarator. As a result, the `numbers` array will have five elements. The `for` loop uses a counter variable named `count`. This loop will iterate five times, and during the loop iterations the `count` variable will take on the values 0 through 4.

Notice that the statement inside the loop uses the `count` variable as a subscript. It assigns 99 to `numbers[count]`. During the first iteration, 99 is assigned to `numbers[0]`. During the next iteration, 99 is assigned to `numbers[1]`. This continues until 99 has been assigned to all of the array's elements. Figure 7-8 illustrates that the loop's initialization, test, and update expressions have been written so the loop starts and ends the counter variable with valid subscript values (0 through 4). This ensures that only valid subscripts are used in the body of the loop.

Figure 7-8



The variable `count` starts at 0, which is the first valid subscript value.

The loop ends when the variable `count` reaches 5, which is the first invalid subscript value.

```
for (count = 0; count < ARRAY_SIZE; count++)
   numbers[count] = 99;
```

The variable `count` is incremented after each iteration.

Program 7-1 could be simplified by using two `for` loops: one for inputting the values into the array and another for displaying the contents of the array. This is shown in Program 7-2.

Program 7-2

```
// This program asks for the number of hours worked
// by six employees. It stores the values in an array.
#include <iostream>
using namespace std;

```

```
 6  int main()
 7  {
 8      const int NUM_EMPLOYEES = 6; // Number of employees
 9      int hours[NUM_EMPLOYEES];    // Each employee's hours
10      int count;                   // Loop counter
11
12      // Input the hours worked.
13      for (count = 0; count < NUM_EMPLOYEES; count++)
14      {
15          cout << “Enter the hours worked by employee ”
16               << (count + 1) << ": ";
17          cin >> hours[count];
18       }
19
20       // Display the contents of the array.
21       cout << "The hours you entered are:";
22       for (count = 0; count < NUM_EMPLOYEES; count++)
23          cout << " " << hours[count];
24       cout << endl;
25       return 0;
26  }
```

Program Output with Example Input Shown in Bold

```
Enter the hours worked by employee 1: 20 [Enter]
Enter the hours worked by employee 2: 12 [Enter]
Enter the hours worked by employee 3: 40 [Enter]
Enter the hours worked by employee 4: 30 [Enter]
Enter the hours worked by employee 5: 30 [Enter]
Enter the hours worked by employee 6: 15 [Enter]
The hours you entered are: 20 12 40 30 30 15
```

The first `for` loop, in lines 13 through 18, prompts the user for each employee's hours. Take a closer look at lines 15 through 17:

```
cout << "Enter the hours worked by employee "
     << (count + 1) << ": ";
cin >> hours[count];
```

Notice that the `cout` statement uses the expression `count + 1` to display the employee number, but the `cin` statement uses `count` as the array subscript. This is because the hours for employee number 1 are stored in `hours[0]`, the hours for employee number 2 are stored in `hours[1]`, and so forth.

The loop in lines 22 through 23 also uses the `count` variable to step through the array, displaying each element.

NOTE: You can use any integer expression as an array subscript. For example, the first loop in Program 7-2 could have been written like this:

```
for (count = 1; count <= NUM_EMPLOYEES; count++)
{
   cout << "Enter the hours worked by employee "
        << count << ": ";
   cin >> hours[count - 1];
}
```

In this code the `cin` statement uses the expression `count - 1` as a subscript.

Inputting data into an array must normally be done one element at a time. For example, the following `cin` statement will not input data into the `hours` array:

```
cin >> hours;   // Wrong! This will NOT work.
```

Instead, you must use multiple `cin` statements to read data into each array element, or use a loop to step through the array, reading data into its elements. Also, outputting an array's contents must normally be done one element at a time. For example, the following `cout` statement will not display the contents of the `hours` array:

```
cout << hours;   // Wrong! This will NOT work.
```

Instead, you must output each element of the array separately.

Reading Data from a File into an Array

Reading the contents of a file into an array is straightforward: Open the file and use a loop to read each item from the file, storing each item in an array element. The loop should iterate until either the array is filled or the end of the file is reached. Program 7-3 demonstrates by opening a file that has 10 numbers stored in it and then reading the file's contents into an array.

Program 7-3

```
// This program reads data from a file into an array.
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    const int ARRAY_SIZE = 10; // Array size
    int numbers[ARRAY_SIZE];    // Array with 10 elements
    int count = 0;              // Loop counter variable
    ifstream inputFile;         // Input file stream object

    // Open the file.
    inputFile.open("TenNumbers.txt");

16      // Read the numbers from the file into the array.
17      while (count < ARRAY_SIZE && inputFile >> numbers[count])
18          count++;
19
20      // Close the file.
21      inputFile.close();
22
23      // Display the numbers read:
24      cout << "The numbers are: ";
25      for (count = 0; count < ARRAY_SIZE; count++)
26          cout << numbers[count] << " ";
27      cout << endl;
28      return 0;
29  }
```

Program Output

```
The numbers are: 101 102 103 104 105 106 107 108 109 110
```

The `while` loop in lines 17 and 18 reads items from the file and assigns them to elements of the `numbers` array. Notice that the loop tests two Boolean expressions, connected by the `&&` operator:

- The first expression is `count < ARRAY_SIZE`. The purpose of this expression is to prevent the loop from writing beyond the end of the array. If the expression is true, the second Boolean expression is tested. If this expression is false, however, the loop stops.
- The second expression is `inputFile >> numbers[count]`. This expression reads a value from the file and stores it in the `numbers[count]` array element. If a value is successfully read from the file, the expression is true and the loop continues. If no value can be read from the file, however, the expression is false and the loop stops.

Each time the loop iterates, it increments `count` in line 18.

Writing the Contents of an Array to a File

Writing the contents of an array to a file is also a straightforward matter. Use a loop to step through each element of the array, writing its contents to a file. Program 7-4 demonstrates.

Program 7-4

```
// This program writes the contents of an array to a file.
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    const int ARRAY_SIZE = 10;   // Array size
    int numbers[ARRAY_SIZE];     // Array with 10 elements
    int count;                   // Loop counter variable
    ofstream outputFile;         // Output file stream object

```

(program continues)

Program 7-4 *(continued)*

```
    // Store values in the array.
    for (count = 0; count < ARRAY_SIZE; count++)
       numbers[count] = count;

    // Open a file for output.
    outputFile.open("SavedNumbers.txt");

    // Write the array contents to the file.
    for (count = 0; count < ARRAY_SIZE; count++)
       outputFile << numbers[count] << endl;

    // Close the file.
    outputFile.close();

    // That's it!
    cout << "The numbers were saved to the file.\n ";
    return 0;
}
```

Program Output

```
The numbers were saved to the file.
```

Contents of the File `SavedNumbers.txt`

```
0
1
2
3
4
5
6
7
8
9
```

7.3 No Bounds Checking in C++

CONCEPT: C++ does not prevent you from overwriting an array's bounds.

C++ is a popular language for software developers who have to write fast, efficient code. To increase runtime efficiency, C++ does not provide many of the common safeguards to prevent unsafe memory access found in other languages. For example, C++ does not perform array bounds checking. This means you can write programs with subscripts that go beyond the boundaries of a particular array. Program 7-5 demonstrates this capability.

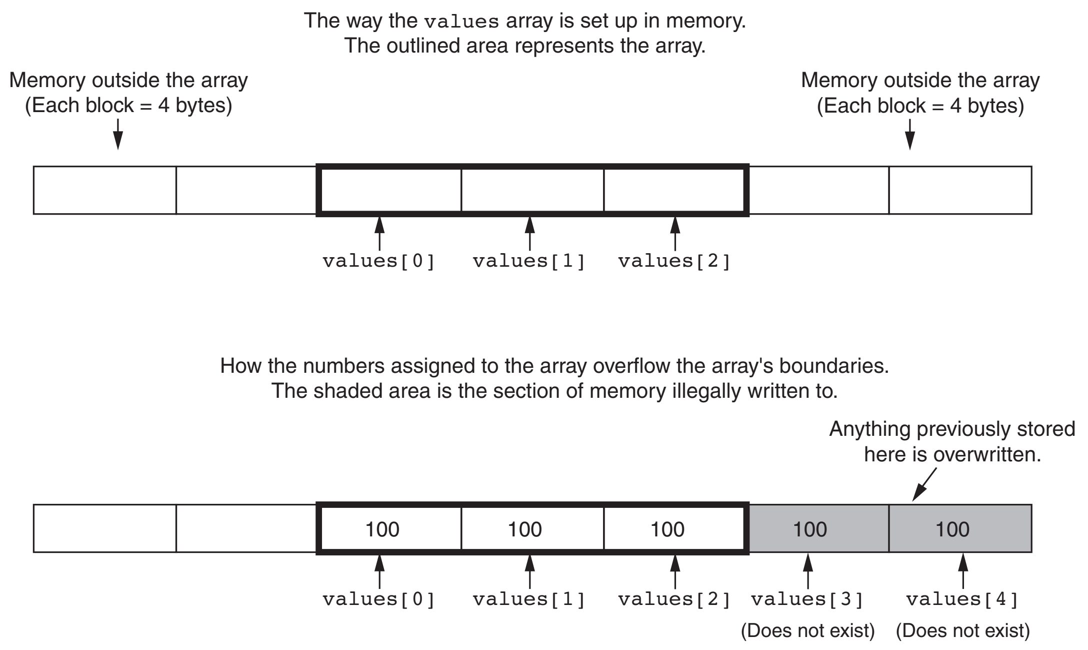
WARNING! Think twice before you compile and run Program 7-5. The program will attempt to write to an area of memory outside the array. This is an invalid operation and will most likely cause the program to crash.

Program 7-5

```
// This program unsafely accesses an area of memory by writing
// values beyond an array's boundary.
// WARNING: If you compile and run this program, it could crash.
#include <iostream>
using namespace std;

int main()
{
    const int SIZE = 3;   // Constant for the array size
    int values[SIZE];     // An array of 3 integers
    int count;            // Loop counter variable

    // Attempt to store five numbers in the three-element array.
    cout << "I will store 5 numbers in a 3-element array!\n";
    for (count = 0; count < 5; count++)
        values[count] = 100;

    // If the program is still running, display the numbers.
    cout << "If you see this message, it means the program\n";
    cout << "has not crashed! Here are the numbers:\n";
    for (count = 0; count < 5; count++)
        cout << values[count] << endl;
    return 0;
}
```

The `values` array has three integer elements, with the subscripts 0, 1, and 2. The loop, however, stores the number 100 in elements 0, 1, 2, 3, and 4. The elements with subscripts 3 and 4 do not exist, but C++ allows the program to write beyond the boundary of the array, as if those elements were there. Figure 7-9 depicts the way the array is set up in memory when the program first starts to execute, and what happens when the loop writes data beyond the boundary of the array.

Figure 7-9


The way the values array is set up in memory.
The outlined area represents the array.
Memory outside the array
(Each block = 4 bytes)
Memory outside the array
(Each block = 4 bytes)
values[0]
values[1]
values[2]
How the numbers assigned to the array overflow the array's boundaries.
The shaded area is the section of memory illegally written to.
Anything previously stored
here is overwritten.
100
100
100
100
100
values[0]
values[1]
values[2]
values[3]
(Does not exist)
values[4]
(Does not exist)

Although C++ programs are fast and efficient, the absence of safeguards such as array bounds checking usually proves to be a bad thing. It's easy for C++ programmers to make careless mistakes that allow programs to access areas of memory that are supposed to be off-limits. You must always make sure that any time you assign values to array elements, the values are written within the array's boundaries.

Watch for Off-by-One Errors

In working with arrays, a common type of mistake is the *off-by-one error*. This is an easy mistake to make because array subscripts start at 0 rather than 1. For example, look at the following code:

```
// This code has an off-by-one error.
const int SIZE = 100;
int numbers[SIZE];
for (int count = 1; count <= SIZE; count++)
   numbers[count] = 0;
```

The intent of this code is to create an array of integers with 100 elements, and store the value 0 in each element. However, this code has an off-by-one error. The loop uses its counter variable, `count`, as a subscript with the `numbers` array. During the loop's execution, the variable `count` takes on the values 1 through 100, when it should take on the values 0 through 99. As a result, the first element, which is at subscript 0, is skipped. In addition, the loop attempts to use 100 as a subscript during the last iteration. Because 100 is an invalid subscript, the program will write data beyond the array's boundaries.

Checkpoint

7.1 Define the following arrays:

A) `empNums`, a 100-element array of `int`s
B) `payRates`, a 25-element array of `float`s
C) `miles`, a 14-element array of `long`s
D) `cityName`, a 26-element array of `string` objects
E) `lightYears`, a 1,000-element array of `double`s

7.2 What's wrong with the following array definitions?

```
int readings[-1];
float measurements[4.5];
int size;
string names[size];
```

7.3 What would the valid subscript values be in a four-element array of `double`s?

7.4 What is the difference between an array's size declarator and a subscript?

7.5 What is "array bounds checking"? Does C++ perform it?

7.6 What is the output of the following code?

```
int values[5], count;
for (count = 0; count < 5; count++)
    values[count] = count + 1;
for (count = 0; count < 5; count++)
    cout << values[count] << endl;
```

7.7 The following program skeleton contains a 20-element array of `int`s called `fish`. When completed, the program should ask how many fish were caught by fishermen 1 through 20, and store this data in the array. Complete the program.

```
#include <iostream>
using namespace std;
int main()
{
   const int NUM_FISH = 20;
   int fish[NUM_FISH];
   // You must finish this program. It should ask how
   // many fish were caught by fishermen 1-20, and
   // store this data in the array fish.
   return 0;
}
```

7.4 Array Initialization

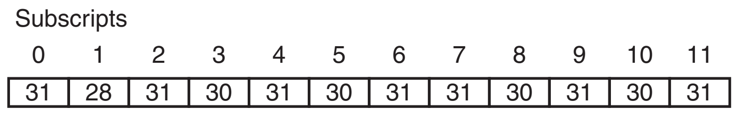
CONCEPT: Arrays may be initialized when they are defined.

Like regular variables, C++ allows you to initialize an array's elements when you create the array. Here is an example:

```
const int MONTHS = 12;
int days[MONTHS] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

The series of values inside the braces and separated with commas is called an *initialization list*. These values are stored in the array elements in the order they appear in the list. (The first value, 31, is stored in `days[0]`, the second value, 28, is stored in `days[1]`, and so forth.) Figure 7-10 shows the contents of the array after the initialization.

Figure 7-10



Program 7-6 demonstrates how an array may be initialized.

Program 7-6

```
// This program displays the number of days in each month.
#include <iostream>
using namespace std;

int main()
{
    const int MONTHS = 12;
    int days[MONTHS] = { 31, 28, 31, 30,
                         31, 30, 31, 31,
                         30, 31, 30, 31};

    for (int count = 0; count < MONTHS; count++)
    {
        cout << "Month " << (count + 1) << " has ";
        cout << days[count] << “ days.\n";
    }
    return 0;
}
```

Program Output

```
Month 1 has 31 days.
Month 2 has 28 days.
Month 3 has 31 days.
Month 4 has 30 days.
Month 5 has 31 days.
Month 6 has 30 days.
Month 7 has 31 days.
Month 8 has 31 days.
Month 9 has 30 days.
Month 10 has 31 days.
Month 11 has 30 days.
Month 12 has 31 days.
```

NOTE: Notice that C++ allows you to spread the initialization list across multiple lines. Both of the following array definitions are equivalent:

```
double coins[5] = {0.05, 0.1, 0.25, 0.5, 1.0};
double coins[5] = {0.05,
                   0.1,
                   0.25,
                   0.5,
                   1.0};
```

Program 7-7 shows an example with a `string` array that is initialized with strings.

Program 7-7

```
// This program initializes a string array.
#include <iostream>
#include <string>
using namespace std;

int main()
{
    const int SIZE = 9;
    string planets[SIZE] = { "Mercury", "Venus", "Earth", "Mars",
                             "Jupiter", "Saturn", "Uranus",
                             "Neptune", "Pluto (a dwarf planet)" };

    cout << "Here are the planets:\n";

    for (int count = 0; count < SIZE; count++)
        cout << planets[count] << endl;
    return 0;
}
```

Program Output

```
Here are the planets:
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
Pluto (a dwarf planet)
```

Program 7-8 shows a character array being initialized with the first ten letters of the alphabet. The array is then used to display those characters' ASCII codes.

Program 7-8

```
// This program uses an array of ten characters to store the
// first ten letters of the alphabet. The ASCII codes of the
// characters are displayed.
#include <iostream>
using namespace std;

int main()
{
    const int NUM_LETTERS = 10;
    char letters[NUM_LETTERS] = {'A', 'B', 'C', 'D', 'E',
                                 'F', 'G', 'H', 'I', 'J'};

    cout << "Character" << "\t" << "ASCII Code\n";
    cout << "---------" << "\t" << "----------\n";
    for (int count = 0; count < NUM_LETTERS; count++)
    {
        cout << letters[count] << "\t\t";
        cout << static_cast<int>(letters[count]) << endl;
    }
    return 0;
}
```

Program Output

```
Character     ASCII Code
---------     ----------
A             65
B             66
C             67
D             68
E             69
F             70
G             71
H             72
I             73
J             74
```

NOTE: An array's initialization list cannot have more values than the array has elements.

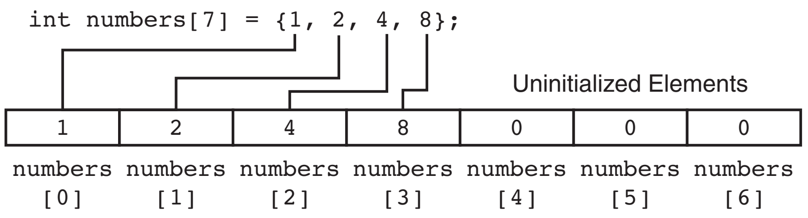
Partial Array Initialization

When an array is being initialized, C++ does not require a value for every element. It's possible to only initialize part of an array, such as:

```
int numbers[7] = {1, 2, 4, 8};
```

This definition initializes only the first four elements of a seven-element array, as illustrated in Figure 7-11.

Figure 7-11



It's important to note that if an array is partially initialized, the uninitialized elements will be set to zero. The uninitialized elements of a `string` array will contain empty strings. This is true even if the array is defined locally. (If a local array is completely uninitialized, its elements will contain "garbage," like all other local variables.) Program 7-9 shows the contents of the array `numbers` after it is partially initialized.

Program 7-9

```
// This program has a partially initialized array.
#include <iostream>
using namespace std;

int main()
{
    const int SIZE = 7;
    int numbers[SIZE] = {1, 2, 4, 8}; // Initialize first 4 elements

    cout << "Here are the contents of the array:\n";
    for (int index = 0; index < SIZE; index++)
       cout << numbers[index] << " ";

    cout << endl;
    return 0;
}
```

Program Output

```
Here are the contents of the array:
1 2 4 8 0 0 0
```

If you leave an element uninitialized, you must leave all the elements that follow it uninitialized as well. C++ does not provide a way to skip elements in the initialization list. For example, the following is *not* legal:

```
int numbers[6] = {2, 4, , 8, , 12}; // NOT Legal!
```

Implicit Array Sizing

It's possible to define an array without specifying its size, as long as you provide an initialization list. C++ automatically makes the array large enough to hold all the initialization values. For example, the following definition creates an array with five elements:

```
double ratings[] = {1.0, 1.5, 2.0, 2.5, 3.0};
```

Because the size declarator is omitted, C++ counts the number of items in the initialization list and gives the array that many elements.

NOTE: You *must* provide an initialization list if you leave out an array's size declarator. Otherwise, C++ doesn't know how large to make the array.

11 7.5 The Range-Based `for` Loop

CONCEPT: The range-based `for` loop is a loop that iterates once for each element in an array. Each time the loop iterates, it copies an element from the array to a variable. The range-based `for` loop was introduced in C++ 11.

C++ 11 provides a specialized version of the `for` loop that, in many circumstances, simplifies array processing. It is known as the *range-based `for` loop*. When you use the range-based `for` loop with an array, the loop automatically iterates once for each element in the array. For example, if you use the range-based `for` loop with an eight-element array, the loop will iterate eight times. Because the range-based `for` loop automatically knows the number of elements in an array, you do not have to use a counter variable to control its iterations, as with a regular `for` loop. Additionally, you do not have to worry about stepping outside the bounds of an array when you use the range-based `for` loop.

The range-based `for` loop is designed to work with a built-in variable known as the *range variable*. Each time the range-based `for` loop iterates, it copies an array element to the range variable. For example, the first time the loop iterates, the range variable will contain the value of element 0, the second time the loop iterates, the range variable will contain the value of element 1, and so forth.

Here is the general format of the range-based `for` loop:

```
for (dataType rangeVariable : array)
    statement;
```

Let's look at the syntax more closely as follows:

- *`dataType`* is the data type of the range variable. It must be the same as the data type of the array elements, or a type that the elements can automatically be converted to.
- *`rangeVariable`* is the name of the range variable. This variable will receive the value of a different array element during each loop iteration. During the first loop iteration, it receives the value of the first element; during the second iteration, it receives the value of the second element, and so forth.
- *`array`* is the name of an array on which you wish the loop to operate. The loop will iterate once for every element in the array.
- *`statement`* is a statement that executes during a loop iteration. If you need to execute more than one statement in the loop, enclose the statements in a set of braces.

For example, assume that you have the following array definition:

```
int numbers[] = { 3, 6, 9 };
```

You can use the following range-based `for` loop to display the contents of the `numbers` array:

```
for (int val : numbers)
   cout << val << endl;
```

Because the `numbers` array has three elements, this loop will iterate three times. The first time it iterates, the `val` variable will receive the value in `numbers[0]`. During the second iteration, `val` will receive the value in `numbers[1]`. During the third iteration, `val` will receive the value in `numbers[2]`. The code's output will be as follows:

```
3
6
9
```

Here is an example of a range-based `for` loop that executes more than one statement in the body of the loop:

```
int[] numbers = { 3, 6, 9 };
for (int val : numbers)
{
   cout << "The next value is ";
   cout << val << endl;
}
```

This code will produce the following output:

```
The next value is 3
The next value is 6
The next value is 9
```

If you wish, you can use the `auto` key word to specify the range variable's data type. Here is an example:

```
int[] numbers = { 3, 6, 9 };
for (auto val : numbers)
    cout << val << endl;
```

Program 7-10 demonstrates the range-based for loop by displaying the elements of an `int` array.

Program 7-10

```
// This program demonstrates the range-based for loop.
#include <iostream>
using namespace std;

int main()
{
    // Define an array of integers.
    int numbers[] = { 10, 20, 30, 40, 50 };

    // Display the values in the array.
    for (int val : numbers)
        cout << val << endl;

    return 0;
}
```

(program output continues)

Program 7-10 *(continued)*

Program Output

```
10
20
30
40
50
```

Program 7-11 shows another example of the range-based `for` loop. This program displays the elements of a `string` array.

Program 7-11

```cpp
// This program demonstrates the range-based for loop.
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string planets[] = { "Mercury", "Venus", "Earth", "Mars",
                         "Jupiter", "Saturn", "Uranus",
                         "Neptune", "Pluto (a dwarf planet)" };

    cout << "Here are the planets:\n";

    // Display the values in the array.
    for (string val : planets)
        cout << val << endl;

    return 0;
}
```

Program Output

```
Here are the planets:
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
Pluto (a dwarf planet)
```

Modifying an Array with a Range-Based `for` Loop

As the range-based `for` loop executes, its range variable contains only a copy of an array element. As a consequence, you cannot use a range-based `for` loop to modify the contents of an array unless you declare the range variable as a reference. Recall from Chapter 6 that a reference variable is an alias for another value. Any changes made to the reference variable are actually made to the value for which it is an alias.

To declare the range variable as a reference variable, simply write an ampersand (`&`) in front of its name in the loop header. Program 7-12 shows an example.

Program 7-12

```
 1  // This program uses a range-based for loop to
 2  // modify the contents of an array.
 3  #include <iostream>
 4  using namespace std;
 5
 6  int main()
 7  {
 8      const int SIZE = 5;
 9      int numbers[5];
10
11      // Get values for the array.
12      for (int &val : numbers)
13      {
14          cout << "Enter an integer value: ";
15          cin >> val;
16      }
17
18      // Display the values in the array.
19      cout << "Here are the values you entered:\n";
20      for (int val : numbers)
21          cout << val << endl;
22
23      return 0;
24  }
```

Program Output with Example Input Shown in Bold

```
Enter an integer value: 1 [Enter]
Enter an integer value: 2 [Enter]
Enter an integer value: 3 [Enter]
Enter an integer value: 4 [Enter]
Enter an integer value: 5 [Enter]
Here are the values you entered:
1
2
3
4
5
```

Notice that in line 12 the range variable, `val`, has an ampersand (`&`) written in front of its name. This declares `val` as a reference variable. As the loop executes, the `val` variable will not merely contain a copy of an array element, but it will be an alias for the element. Any changes made to the `val` variable will actually be made to the array element it references.

Also notice that in line 20 we did not declare `val` as a reference variable (there is no ampersand written in front of the variable's name). Because the loop is simply displaying the array elements, and does not need to change the array's contents, there is no need to make `val` a reference variable.

The Range-Based `for` Loop versus the Regular `for` Loop

The range-based `for` loop can be used in any situation where you need to step through the elements of an array, and you do not need to use the element subscripts. It will not work, however, in situations where you need the element subscript for some purpose. In those situations, you need to use the regular `for` loop.

NOTE: You can use the `auto` key word with a reference range variable. For example, the code in lines 12 through 16 in Program 7-12 could have been written like this:

```
for (auto &val : numbers)
{
    cout << "Enter an integer value: ";
    cin >> val;
}
```

7.6 Processing Array Contents

CONCEPT: Individual array elements are processed like any other type of variable.

Processing array elements is no different than processing other variables. For example, the following statement multiplies `hours[3]` by the variable `rate`:

```
pay = hours[3] * rate;
```

And the following are examples of pre-increment and post-increment operations on array elements:

```
int score[5] = {7, 8, 9, 10, 11};
++score[2];    // Pre-increment operation on the value in score[2]
score[4]++;    // Post-increment operation on the value in score[4]
```

NOTE: When using increment and decrement operators, be careful not to confuse the subscript with the array element. For example, the following statement decrements the variable `count`, but does nothing to the value in `amount[count]`:

```
amount[count--];
```

To decrement the value stored in `amount[count]`, use the following statement:

```
amount[count]--;
```

Program 7-13 demonstrates the use of array elements in a simple mathematical statement. A loop steps through each element of the array, using the elements to calculate the gross pay of five employees.

Program 7-13

```
// This program stores, in an array, the hours worked by
// employees who all make the same hourly wage.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    const int NUM_EMPLOYEES = 5;  // Number of employees
    int hours[NUM_EMPLOYEES];     // Array to hold hours
    double payrate;               // Hourly pay rate
    double grossPay;              // To hold the gross pay

    // Input the hours worked.
    cout << "Enter the hours worked by ";
    cout << NUM_EMPLOYEES << " employees who all\n";
    cout << "earn the same hourly rate.\n";
    for (int index = 0; index < NUM_EMPLOYEES; index++)
    {
        cout << "Employee #" << (index + 1) << ": ";
        cin >> hours[index];
    }

    // Input the hourly rate for all employees.
    cout << "Enter the hourly pay rate for all the employees:  ";
    cin >> payrate;

    // Display each employee's gross pay.
    cout << "Here is the gross pay for each employee:\n";
    cout << fixed << showpoint << setprecision(2);
    for (int index = 0; index < NUM_EMPLOYEES; index++)
    {
        grossPay = hours[index] * payrate;
        cout << "Employee #" << (index + 1);
        cout << ": $" << grossPay << endl;
    }
    return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter the hours worked by 5 employees who all
earn the same hourly rate.
Employee #1: 5 [Enter]
Employee #2: 10 [Enter]
```

(program output continues)

Program 7-13 *(continued)*

```
Employee #3: 15 [Enter]
Employee #4: 20 [Enter]
Employee #5: 40 [Enter]
Enter the hourly pay rate for all the employees: 12.75 [Enter]
Here is the gross pay for each employee:
Employee #1: $63.75
Employee #2: $127.50
Employee #3: $191.25
Employee #4: $255.00
Employee #5: $510.00
```

The following statement in line 33 assigns the value of `hours[index]` times `payRate` to the `grossPay` variable:

```
grossPay = hours[index] * payRate;
```

Array elements may also be used in relational expressions. For example, the following `if` statement tests `cost[20]` to determine whether it is less than `cost[0]`:

```
if (cost[20] < cost[0])
```

And the following statement sets up a `while` loop to iterate as long as `value[place]` does not equal 0:

```
while (value[place] != 0)
```

Thou Shall Not Assign

The following code defines two integer arrays: `newValues` and `oldValues`. `newValues` is uninitialized and `oldValues` is initialized with 10, 100, 200, and 300:

```
const int SIZE = 4;
int oldValues[SIZE] = {10, 100, 200, 300};
int newValues[SIZE];
```

At first glance, it might appear that the following statement assigns the contents of the array `oldValues` to `newValues`:

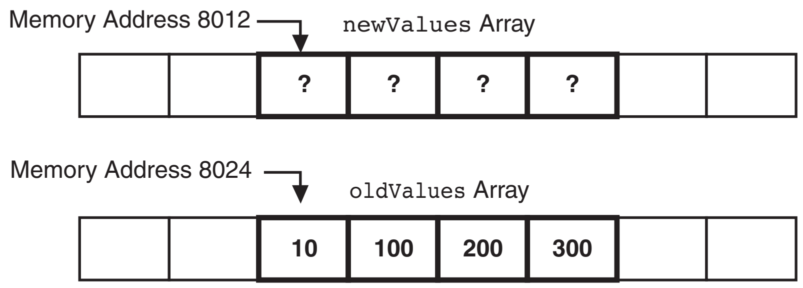
```
newValues = oldValues; // Wrong!
```

Unfortunately, this statement will not work. The only way to assign one array to another is to assign the individual elements in the arrays. Usually, this is best done with a loop, such as:

```
for (int count = 0; count < SIZE; count++)
    newValues[count] = oldValues[count];
```

The reason the assignment operator will not work with an entire array at once is complex, but important to understand. Anytime the name of an array is used without brackets and a subscript, *it is seen as the array's beginning memory address*. To illustrate this, consider the definition of the arrays `newValues` and `oldValues` above. Figure 7-12 depicts the two arrays in memory.

Figure 7-12



In the figure, `newValues` is shown starting at memory address 8012 and `oldValues` is shown starting at 8024. (Of course, these are just arbitrary addresses, picked for illustration purposes. In reality the addresses would probably be different.) Table 7-2 shows various expressions that use the names of these arrays, and their values.

Table 7-2

Expression	Value
`oldValues[0]`	10 (Contents of Element 0 of `oldValues`)
`oldValues[1]`	100 (Contents of Element 1 of `oldValues`)
`oldValues[2]`	200 (Contents of Element 2 of `oldValues`)
`oldValues[3]`	300 (Contents of Element 3 of `oldValues`)
`newValues`	8012 (Memory Address of `newValues`)
`oldValues`	8024 (Memory Address of `oldValues`)

Because the name of an array without the brackets and subscript stands for the array's starting memory address, the following statement

```
newValues = oldValues;
```

is interpreted by C++ as

```
8012 = 8024;
```

The statement will not work because you cannot change the starting memory address of an array.

Printing the Contents of an Array

Suppose we have the following array definition:

```
const int SIZE = 5;
int numbers [SIZE] = {10, 20, 30, 40, 50};
```

You now know that an array's name is seen as the array's beginning memory address. This explains why the following statement cannot be used to display the contents of `array`:

```
cout << numbers << endl;  //Wrong!
```

When this statement executes, `cout` will display the array's memory address, not the array's contents. You must use a loop to display the contents of each of the array's elements, as follows.

```
for (int count = 0; count < SIZE; count++)
    cout << numbers[count] << endl;
```

11 In C++ 11, you can use the range-based `for` loop to display an array's contents, as shown here:

```
for (int val : numbers)
    cout << val << endl;
```

Summing the Values in a Numeric Array

To sum the values in an array, you must use a loop with an accumulator variable. The loop adds the value in each array element to the accumulator. For example, assume that the following statements appear in a program and that values have been stored in the `units` array.

```
const int NUM_UNITS = 24;
int units[NUM_UNITS];
```

The following code uses a regular `for` loop to add the values of each element in the array to the `total` variable. When the code is finished, `total` will contain the sum of the `units` array's elements.

```
int total = 0;   // Initialize accumulator
for (int count = 0; count < NUM_UNITS; count++)
    total += units[count];
```

11 In C++ 11, you can use the range-based `for` loop, as shown here. When the code is finished, `total` will contain the sum of the `units` array's elements.

```
int total = 0;   // Initialize accumulator
for (int val : units)
    total += val;
```

NOTE: The first statement in both of these code segments sets `total` to 0. Recall from Chapter 5 that an accumulator variable must be set to 0 before it is used to keep a running total or the sum will not be correct.

Getting the Average of the Values in a Numeric Array

The first step in calculating the average of all the values in an array is to sum the values. The second step is to divide the sum by the number of elements in the array. Assume that the following statements appear in a program and that values have been stored in the `scores` array.

```
const int NUM_SCORES = 10;
double scores[NUM_SCORES];
```

The following code calculates the average of the values in the `scores` array. When the code completes, the average will be stored in the `average` variable.

```
double total = 0;    // Initialize accumulator
double average;      // Will hold the average
for (int count = 0; count < NUM_SCORES; count++)
    total += scores[count];
average = total / NUM_SCORES;
```

Notice that the last statement, which divides `total` by `numScores`, is not inside the loop. This statement should only execute once, after the loop has finished its iterations.

11 In C++ 11, you can use the range-based `for` loop, as shown here. When the code completes, the average will be stored in the `average` variable.

```
double total = 0;  // Initialize accumulator
double average;    // Will hold the average
for (int val : scores)
     total += val;
average = total / NUM_SCORES;
```

Finding the Highest and Lowest Values in a Numeric Array

The algorithms for finding the highest and lowest values in an array are very similar. First, let's look at code for finding the highest value in an array. Assume that the following code exists in a program, and that values have already been stored in the `numbers` array.

```
const int SIZE = 50;
int numbers[SIZE];
```

The code to find the highest value in the array is as follows.

```
int count;
int highest;

highest = numbers[0];
for (count = 1; count < SIZE; count++)
{
   if (numbers[count] > highest)
      highest = numbers[count];
}
```

First we copy the value in the first array element to the variable `highest`. Then the loop compares all of the remaining array elements, beginning at subscript 1, to the value in `highest`. Each time it finds a value in the array that is greater than `highest`, it copies that value to `highest`. When the loop has finished, `highest` will contain the highest value in the array.

The following code finds the lowest value in the array. As you can see, it is nearly identical to the code for finding the highest value.

```
int count;
int lowest;

lowest = numbers[0];
for (count = 1; count < SIZE; count++)
{
   if (numbers[count] < lowest)
      lowest = numbers[count];
}
```

When the loop has finished, `lowest` will contain the lowest value in the array.

Partially Filled Arrays

Sometimes you need to store a series of items in an array, but you do not know the number of items that there are. As a result, you do not know the exact number of elements needed for the array. One solution is to make the array large enough to hold the largest possible number of items. This can lead to another problem, however. If the actual number of items stored in the array is less than the number of elements, the array will be only partially filled. When you process a partially filled array, you must only process the elements that contain valid data items.

A partially filled array is normally used with an accompanying integer variable that holds the number of items stored in the array. For example, suppose a program uses the following code to create an array with 100 elements, and an `int` variable named `count` that will hold the number of items stored in the array:

```
const int SIZE = 100;
int numbers[SIZE];
int count = 0;
```

Each time we add an item to the array, we must increment `count`. The following code demonstrates.

```
int num;
cout << "Enter a number or -1 to quit: ";
cin >> num;
while (num != -1 && count < SIZE)
{
   count++;
   numbers[count - 1] = num;
   cout << "Enter a number or -1 to quit: ";
   cin >> num;
}
```

Each iteration of this sentinel-controlled loop allows the user to enter a number to be stored in the array, or −1 to quit. The `count` variable is incremented and then used to calculate the subscript of the next available element in the array. When the user enters −1, or `count` exceeds 99, the loop stops. The following code displays all of the valid items in the array.

```
for (int index = 0; index < count; index++)
{
   cout << numbers[index] << endl;
}
```

Notice that this code uses `count` to determine the maximum array subscript to use.

Program 7-14 shows how this technique can be used to read an unknown number of items from a file into an array. The program reads values from the file numbers.txt.

Program 7-14

```
//This program reads data from a file into an array.
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    const int ARRAY_SIZE = 100; // Array size
    int numbers[ARRAY_SIZE];    // Array with 100 elements
10        int count = 0;                 // Loop counter variable
11        ifstream inputFile;            // Input file stream object
12
13        inputFile.open("numbers.txt"); // Open the file.
14
15        // Read the numbers from the file into the array.
16        // After this loop executes, the count variable will hold
17        // the number of values that were stored in the array.
18        while (count < ARRAY_SIZE && inputFile >> numbers[count])
19            count++;
20
21        // Close the file.
22        inputFile.close();
23
24        // Display the numbers read.
25        cout << "The numbers are: ";
26        for (int index = 0; index < count; index++)
27            cout << numbers[index] << " ";
28        cout << endl;
29        return 0;
30    }
```

Program Output

```
The numbers are: 47 89 65 36 12 25 17 8 62 10 87 62
```

Look closer at the `while` loop that begins in line 18. It repeats as long as `count` is less than `ARRAY_SIZE` and the end of the file has not been encountered. The first part of the `while` loop's test expression, `count < ARRAY_SIZE`, prevents the loop from writing outside the array boundaries. Recall from Chapter 4 that the `&&` operator performs short-circuit evaluation, so the second part of the `while` loop's test expression, `inputFile >> values[count]`, will be executed only if `count` is less than `ARRAY_SIZE`.

Comparing Arrays

We have already noted that you cannot simply assign one array to another array. You must assign each element of the first array to an element of the second array. In addition, you cannot use the == operator with the names of two arrays to determine whether the arrays are equal. For example, the following code appears to compare two arrays, but in reality does not.

```
int firstArray[] = { 5, 10, 15, 20, 25 };
int secondArray[] = { 5, 10, 15, 20, 25 };
if (firstArray == secondArray)       // This is a mistake.
   cout << "The arrays are the same.\n";
else
   cout << "The arrays are not the same.\n";
```

When you use the == operator with array names, the operator compares the beginning memory addresses of the arrays, not the contents of the arrays. The two array names in this code will obviously have different memory addresses. Therefore, the result of the expression `firstArray == secondArray` is false and the code reports that the arrays are not the same.

To compare the contents of two arrays, you must compare the elements of the two arrays. For example, look at the following code.

```
const int SIZE = 5;
int firstArray[SIZE] = { 5, 10, 15, 20, 25 };
int secondArray[SIZE] = { 5, 10, 15, 20, 25 };
bool arraysEqual = true; // Flag variable
int count = 0;           // Loop counter variable

// Determine whether the elements contain the same data.
while (arraysEqual && count < SIZE)
{
   if (firstArray[count] != secondArray[count])
      arraysEqual = false;
   count++;
}

if (arraysEqual)
   cout << "The arrays are equal.\n";
else
   cout << "The arrays are not equal.\n";
```

This code determines whether `firstArray` and `secondArray` contain the same values. A `bool` variable, `arraysEqual`, which is initialized to `true`, is used to signal whether the arrays are equal. Another variable, `count`, which is initialized to 0, is used as a loop counter variable.

Then a `while` loop begins. The loop executes as long as `arraysEqual` is `true` and the counter variable `count` is less than `SIZE`. During each iteration, it compares a different set of corresponding elements in the arrays. When it finds two corresponding elements that have different values, the `arraysEqual` variable is set to `false`. After the loop finishes, an `if` statement examines the `arraysEqual` variable. If the variable is `true`, then the arrays are equal and a message indicating so is displayed. Otherwise, they are not equal, so a different message is displayed.

7.7 Focus on Software Engineering: Using Parallel Arrays

CONCEPT: By using the same subscript, you can build relationships between data stored in two or more arrays.

Sometimes it's useful to store related data in two or more arrays. It's especially useful when the related data is of unlike types. For example, Program 7-15 is another variation of the payroll program. It uses two arrays: one to store the hours worked by each employee (as `int`s), and another to store each employee's hourly pay rate (as `double`s).

Program 7-15

```
// This program uses two parallel arrays: one for hours
// worked and one for pay rate.
#include <iostream>
```

```
#include <iomanip>
using namespace std;

int main()
{
    const int NUM_EMPLOYEES = 5;   // Number of employees
    int hours[NUM_EMPLOYEES];      // Holds hours worked
    double payRate[NUM_EMPLOYEES]; // Holds pay rates

    // Input the hours worked and the hourly pay rate.
    cout << "Enter the hours worked by " << NUM_EMPLOYEES
         << " employees and their\n"
         << "hourly pay rates.\n";
    for (int index = 0; index < NUM_EMPLOYEES; index++)
    {
        cout << "Hours worked by employee #" << (index+1) << ": ";
        cin >> hours[index];
        cout << "Hourly pay rate for employee #" << (index+1) << ": ";
        cin >> payRate[index];
    }

    // Display each employee's gross pay.
    cout << "Here is the gross pay for each employee:\n";
    cout << fixed << showpoint << setprecision(2);
    for (int index = 0; index < NUM_EMPLOYEES; index++)
    {
        double grossPay = hours[index] * payRate[index];
        cout << "Employee #" << (index + 1);
        cout << ": $" << grossPay << endl;
    }
    return 0;
}
```

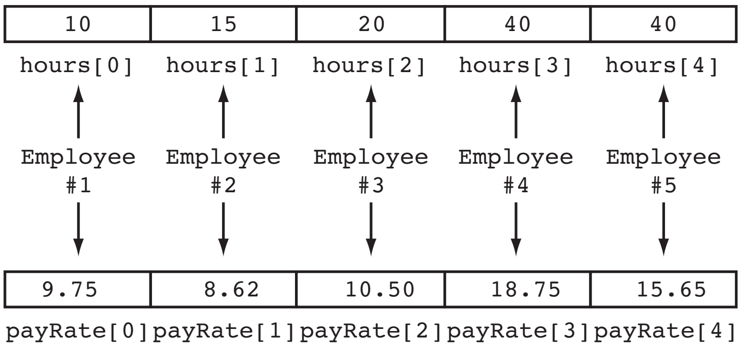
Program Output with Example Input Shown in Bold

```
Enter the hours worked by 5 employees and their
hourly pay rates.
Hours worked by employee #1: 10 [Enter]
Hourly pay rate for employee #1: 9.75 [Enter]
Hours worked by employee #2: 15 [Enter]
Hourly pay rate for employee #2: 8.62 [Enter]
Hours worked by employee #3: 20 [Enter]
Hourly pay rate for employee #3: 10.50 [Enter]
Hours worked by employee #4: 40 [Enter]
Hourly pay rate for employee #4: 18.75 [Enter]
Hours worked by employee #5: 40 [Enter]
Hourly pay rate for employee #5: 15.65 [Enter]
Here is the gross pay for each employee:
Employee #1: $97.50
Employee #2: $129.30
Employee #3: $210.00
Employee #4: $750.00
Employee #5: $626.00
```

Notice in the loops that the same subscript is used to access both arrays. That's because the data for one employee is stored in the same relative position in each array. For example, the hours worked by employee #1 are stored in `hours[0]`, and the same employee's pay rate is stored in `payRate[0]`. The subscript relates the data in both arrays.

This concept is illustrated in Figure 7-13.

Figure 7-13



Checkpoint

7.8 Define the following arrays:

A) `ages`, a 10-element array of `int`s initialized with the values 5, 7, 9, 14, 15, 17, 18, 19, 21, and 23.

B) `temps`, a 7-element array of `float`s initialized with the values 14.7, 16.3, 18.43, 21.09, 17.9, 18.76, and 26.7.

C) `alpha`, an 8-element array of `char`s initialized with the values 'J', 'B', 'L', 'A', '*', '$', 'H', and 'M'.

7.9 Is each of the following a valid or invalid array definition? (If a definition is invalid, explain why.)

```
int numbers[10] = {0, 0, 1, 0, 0, 1, 0, 0, 1, 1};
int matrix[5] = {1, 2, 3, 4, 5, 6, 7};
double radii[10] = {3.2, 4.7};
int table[7] = {2, , , 27, , 45, 39};
char codes[] = {'A', 'X', '1', '2', 's'};
int blanks[];
```

7.10 Given the following array definition:

```
int values[] = {2, 6, 10, 14};
```

What does each of the following display?

A) `cout << values[2];`

B) `cout << ++values[0];`

C) `cout << values[1]++;`

D)
```
x = 2;
cout << values[++x];
```

7.11 Given the following array definition:

```
int nums[5] = {1, 2, 3};
```

What will the following statement display?

```
cout << nums[3];
```

7.12 What is the output of the following code? (You may need to use a calculator.)

```
double balance[5] = {100.0, 250.0, 325.0, 500.0, 1100.0};
const double INTRATE = 0.1;

cout << fixed << showpoint << setprecision(2);
for (int count = 0; count < 5; count++)
    cout << (balance[count] * INTRATE) << endl;
```

7.13 What is the output of the following code? (You may need to use a calculator.)

```
const int SIZE = 5;
int time[SIZE] = {1, 2, 3, 4, 5},
    speed[SIZE] = {18, 4, 27, 52, 100},
    dist[SIZE];

for (int count = 0; count < SIZE; count++)
    dist[count] = time[count] * speed[count];
for (int count = 0; count < SIZE; count++)
{
    cout << time[count] << " ";
    cout << speed[count] << " ";
    cout << dist[count] << endl;
}
```

7.8 Arrays as Function Arguments

CONCEPT: To pass an array as an argument to a function, pass the name of the array.

VideoNote
Passing an Array to a Function

Quite often you'll want to write functions that process the data in arrays. For example, functions could be written to put values in an array, display an array's contents on the screen, total all of an array's elements, or calculate their average. Usually, such functions accept an array as an argument.

When a single element of an array is passed to a function, it is handled like any other variable. For example, Program 7-16 shows a loop that passes one element of the array `numbers` to the function `showValue` each time the loop iterates.

Program 7-16

```
// This program demonstrates that an array element is passed
// to a function like any other variable.
#include <iostream>
using namespace std;

void showValue(int); // Function prototype

```

(program continues)

Program 7-16 *(continued)*

```
 8  int main()
 9  {
10      const int SIZE = 8;
11      int numbers[SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
12
13      for (int index = 0; index < SIZE; index++)
14          showValue(numbers[index]);
15      return 0;
16  }
17
18  //**********************************************
19  // Definition of function showValue.           *
20  // This function accepts an integer argument.  *
21  // The value of the argument is displayed.     *
22  //**********************************************
23
24  void showValue(int num)
25  {
26      cout << num << " ";
27  }
```

Program Output

```
5 10 15 20 25 30 35 40
```

Each time `showValue` is called in line 14, a copy of an array element is passed into the parameter variable `num`. The `showValue` function simply displays the contents of `num` and doesn't work directly with the array element itself. (In other words, the array element is passed by value.)

If the function were written to accept the entire array as an argument, however, the parameter would be set up differently. In the following function definition, the parameter `nums` is followed by an empty set of brackets. This indicates that the argument will be an array, not a single value.

```
void showValues(int nums[], int size)
{
   for (int index = 0; index < size; index++)
       cout << nums[index] << " ";
   cout << endl;
}
```

The reason there is no size declarator inside the brackets of `nums` is because `nums` is not actually an array. It's a special variable that can accept the address of an array. When an entire array is passed to a function, it is not passed by value, but passed by reference. Imagine the CPU time and memory that would be necessary if a copy of a 10,000-element array were created each time it was passed to a function! Instead, only the starting memory address of the array is passed. Program 7-17 shows the function `showValues` in use.

NOTE: Notice that in the function prototype, empty brackets appear after the data type of the array parameter. This indicates that `showValues` accepts the address of an array of integers.

Program 7-17

```
 1  // This program demonstrates an array being passed to a function.
 2  #include <iostream>
 3  using namespace std;
 4
 5  void showValues(int [], int); // Function prototype
 6
 7  int main()
 8  {
 9      const int ARRAY_SIZE = 8;
10      int numbers[ARRAY_SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
11
12      showValues(numbers, ARRAY_SIZE);
13      return 0;
14  }
15
16  //**************************************************
17  // Definition of function showValue.               *
18  // This function accepts an array of integers and  *
19  // the array's size as its arguments. The contents *
20  // of the array are displayed.                     *
21  //**************************************************
22
23  void showValues(int nums[], int size)
24  {
25      for (int index = 0; index < size; index++)
26          cout << nums[index] << " ";
27      cout << endl;
28  }
```

Program Output

```
5 10 15 20 25 30 35 40
```

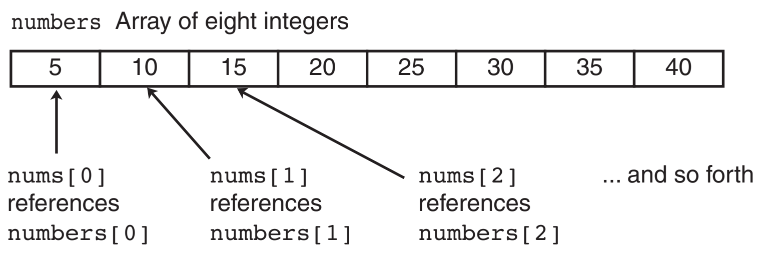
In Program 7-17, the function `showValues` is called in the following statement which appears in line 12:

```
showValues(numbers, ARRAY_SIZE);
```

The first argument is the name of the array. Remember, in C++ the name of an array without brackets and a subscript is actually the beginning address of the array. In this function call, the address of the `numbers` array is being passed as the first argument to the function. The second argument is the size of the array.

In the `showValues` function, the beginning address of the `numbers` array is copied into the `nums` parameter variable. The `nums` variable is then used to reference the `numbers` array. Figure 7-14 illustrates the relationship between the `numbers` array and the `nums` parameter variable. When the contents of `nums[0]` is displayed, it is actually the contents of `numbers[0]` that appears on the screen.

Figure 7-14


numbers Array of eight integers
5
10
15
20
25
30
35
40
nums[0] references numbers[0]
nums[1] references numbers[1]
nums[2] references numbers[2]
... and so forth

NOTE: Although `nums` is not a reference variable, it works like one.

The `nums` parameter variable in the `showValues` function can accept the address of any integer array and can be used to reference that array. So, we can use the `showValues` function to display the contents of any integer array by passing the name of the array and its size as arguments. Program 7-18 uses the function to display the contents of two different arrays.

Program 7-18

```
// This program demonstrates the showValues function being
// used to display the contents of two arrays.
#include <iostream>
using namespace std;

void showValues(int [], int); // Function prototype

int main()
{
    const int SIZE1 = 8; // Size of set1 array
    const int SIZE2 = 5; // Size of set2 array
    int set1[SIZE1] = {5, 10, 15, 20, 25, 30, 35, 40};
    int set2[SIZE2] = {2, 4, 6, 8, 10};

    // Pass set1 to showValues.
    showValues(set1, SIZE1);

    // Pass set2 to showValues.
    showValues(set2, SIZE2);
    return 0;
}

//*****************************************************
// Definition of function showValues.                 *
// This function accepts an array of integers and     *
// the array's size as its arguments. The contents    *
// of the array are displayed.                        *
//*****************************************************

```

```
void showValues(int nums[], int size)
{
    for (int index = 0; index < size; index++)
        cout << nums[index] << " ";
    cout << endl;
}
```

Program Output

```
5 10 15 20 25 30 35 40
2 4 6 8 10
```

Recall from Chapter 6 that when a reference variable is used as a parameter, it gives the function access to the original argument. Any changes made to the reference variable are actually performed on the argument referenced by the variable. Array parameters work very much like reference variables. They give the function direct access to the original array. Any changes made with the array parameter are actually made on the original array used as the argument. The function `doubleArray` in Program 7-19 uses this capability to double the contents of each element in the array.

Program 7-19

```
// This program uses a function to double the value of
// each element of an array.
#include <iostream>
using namespace std;

// Function prototypes
void doubleArray(int [], int);
void showValues(int [], int);

int main()
{
    const int ARRAY_SIZE = 7;
    int set[ARRAY_SIZE] = {1, 2, 3, 4, 5, 6, 7};

    // Display the initial values.
    cout << "The array's values are:\n";
    showValues(set, ARRAY_SIZE);

    // Double the values in the array.
    doubleArray(set, ARRAY_SIZE);

    // Display the resulting values.
    cout << "After calling doubleArray the values are:\n";
    showValues(set, ARRAY_SIZE);

    return 0;
}

```

(program continues)

Program 7-19 *(continued)*

```
//****************************************************
// Definition of function doubleArray                *
// This function doubles the value of each element   *
// in the array passed into nums. The value passed   *
// into size is the number of elements in the array. *
//****************************************************

void doubleArray(int nums[], int size)
{
    for (int index = 0; index < size; index++)
        nums[index] *= 2;
}

//**************************************************
// Definition of function showValues.              *
// This function accepts an array of integers and  *
// the array's size as its arguments. The contents *
// of the array are displayed.                     *
//**************************************************

void showValues(int nums[], int size)
{
    for (int index = 0; index < size; index++)
        cout << nums[index] << " ";
    cout << endl;
}
```

Program Output

```
The array's values are:
1 2 3 4 5 6 7
After calling doubleArray the values are:
2 4 6 8 10 12 14
```

Using `const` Array Parameters

Sometimes you want a function to be able to modify the contents of an array that is passed to it as an argument, and sometimes you don't. You can prevent a function from making changes to an array argument by using the `const` key word in the parameter declaration. Here is an example of the `showValues` function, shown previously, rewritten with a `const` array parameter:

```
void showValues(const int nums[], int size)
{
    for (int index = 0; index < size; index++)
        cout << nums[index] << " ";
    cout << endl;
}
```

When an array parameter is declared as `const`, the function is not allowed to make changes to the array's contents. If a statement in the function attempts to modify the array, an error will occur at compile time. As a precaution, you should always use `const` array parameters in any function that is not intended to modify its array argument. That way, the function will fail to compile if you inadvertently write code in it that modifies the array.

Some Useful Array Functions

Section 7.6 introduced you to algorithms such as summing an array and finding the highest and lowest values in an array. Now that you know how to pass an array as an argument to a function, you can write general purpose functions that perform those operations. The following In the Spotlight section shows an example.

In the Spotlight:
Processing an Array

Dr. LaClaire gives four exams during the semester in her chemistry class. At the end of the semester she drops each student's lowest test score before averaging the scores. She has asked you to write a program that will read a student's four test scores as input, and calculate the average with the lowest score dropped. Here is the pseudocode algorithm that you developed:

```
Read the student's four test scores.
Calculate the total of the scores.
Find the lowest score.
Subtract the lowest score from the total. This gives the adjusted total.
Divide the adjusted total by 3. This is the average.
Display the average.
```

Program 7-20 shows the program, which is modularized. Rather than presenting the entire program at once, let's first examine the `main` function, and then each additional function separately. Here is the first part of the program, including the `main` function:

Program 7-20 **(`main` function)**

```
// This program gets a series of test scores and
// calculates the average of the scores with the
// lowest score dropped.
#include <iostream>
#include <iomanip>
using namespace std;

// Function prototypes
void getTestScores(double[], int);
double getTotal(const double[], int);
double getLowest(const double[], int);

```

(program continues)

Program 7-20 *(continued)*

```
13 int main()
14 {
15     const int SIZE = 4;      // Array size
16     double testScores[SIZE], // Array of test scores
17            total,            // Total of the scores
18            lowestScore,      // Lowest test score
19            average;          // Average test score
20
21     // Set up numeric output formatting.
22     cout << fixed << showpoint << setprecision(1);
23
24     // Get the test scores from the user.
25     getTestScores(testScores, SIZE);
26
27     // Get the total of the test scores.
28     total = getTotal(testScores, SIZE);
29
30     // Get the lowest test score.
31     lowestScore = getLowest(testScores, SIZE);
32
33     // Subtract the lowest score from the total.
34     total -= lowestScore;
35
36     // Calculate the average. Divide by 3 because
37     // the lowest test score was dropped.
38     average = total / (SIZE - 1);
39
40     // Display the average.
41     cout << "The average with the lowest score "
42          << "dropped is " << average << ".\n";
43
44     return 0;
45 }
46
```

Lines 15 through 19 define the following items:

- `SIZE`, an `int` constant that is used as an array size declarator
- `testScores`, a `double` array to hold the test scores
- `tOtal`, a `double` variable that will hold the test score totals
- `lowestScore`, a `double` variable that will hold the lowest test score
- `average`, a `double` variable that will hold the average of the test scores

Line 25 calls the `getTestScores` function, passing the `testScores` array and the value of the `SIZE` constant as arguments. The function gets the test scores from the user and stores them in the array.

Line 28 calls the `getTotal` function, passing the `testScores` array and the value of the `SIZE` constant as arguments. The function returns the total of the values in the array. This value is assigned to the `total` variable.

Line 31 calls the getLowest function, passing the testScores array and the value of the SIZE constant as arguments. The function returns the lowest value in the array. This value is assigned to the lowestScore variable.

Line 34 subtracts the lowest test score from the total variable. Then, line 38 calculates the average by dividing total by SIZE − 1. (The program divides by SIZE − 1 because the lowest test score was dropped.) Lines 41 and 42 display the average.

The getTestScores function appears next, as shown here:

Program 7-20 **(getTestScores function)**

```
47 //************************************************************
48 // The getTestScores function accepts an array and its size *
49 // as arguments. It prompts the user to enter test scores,  *
50 // which are stored in the array.                           *
51 //************************************************************
52 
53 void getTestScores(double scores[], int size)
54 {
55     // Loop counter
56     int index;
57 
58     // Get each test score.
59     for(index = 0; index <= size - 1; index++)
60     {
61         cout << "Enter test score number "
62              << (index + 1) << ": ";
63         cin >> scores[index];
64     }
65 }
66 
```

The getTestScores function has two parameters:

- scores[]—A double array
- size—An int specifying the size of the array that is passed into the scores[] parameter

The purpose of this function is to get a student's test scores from the user and store them in the array that is passed as an argument into the scores[] parameter.

The getTotal function appears next, as shown here:

Program 7-20 **(getTotal function)**

```
67 //******************************************************
68 // The getTotal function accepts a double array       *
69 // and its size as arguments. The sum of the array's *
70 // elements is returned as a double.                  *
71 //******************************************************
72 
```

(program continues)

Program 7-20 *(continued)*

```
double getTotal(const double numbers[], int size)
{
    double total = 0; // Accumulator

    // Add each element to total.
    for (int count = 0; count < size; count++)
        total += numbers[count];

    // Return the total.
    return total;
}

```

The `getTotal` function has two parameters:

- `numbers[]` —A `const double` array
- `size` —An `int` specifying the size of the array that is passed into the `numbers[]` parameter

This function returns the total of the values in the array that is passed as an argument into the `numbers[]` parameter.

The `getLowest` function appears next, as shown here:

Program 7-20 **(`getLowest` function)**

```
//*****************************************************
// The getLowest function accepts a double array and *
// its size as arguments. The lowest value in the    *
// array is returned as a double.                    *
//*****************************************************

double getLowest(const double numbers[], int size)
{
    double lowest; // To hold the lowest value

    // Get the first array's first element.
    lowest = numbers[0];

    // Step through the rest of the array. When a
    // value less than lowest is found, assign it
    // to lowest.
    for (int count = 1; count < size; count++)
    {
        if (numbers[count] < lowest)
            lowest = numbers[count];
    }

    // Return the lowest value.
    return lowest;
}
```

The `getLowest` function has two parameters:

- `numbers[]`—A `const double` array
- `size`—An `int` specifying the size of the array that is passed into the `numbers[]` parameter

This function returns the lowest value in the array that is passed as an argument into the `numbers[]` parameter. Here is an example of the program's output:

Program 7-20

Program Output with Example Input Shown in Bold

```
Enter test score number 1: 92 [Enter]
Enter test score number 2: 67 [Enter]
Enter test score number 3: 75 [Enter]
Enter test score number 4: 88 [Enter]
The average with the lowest score dropped is 85.0.
```

Checkpoint

7.14 Given the following array definitions

```
double array1[4] = {1.2, 3.2, 4.2, 5.2};
double array2[4];
```

will the following statement work? If not, why?

```
array2 = array1;
```

7.15 When an array name is passed to a function, what is actually being passed?

7.16 When used as function arguments, are arrays passed by value?

7.17 What is the output of the following program? (You may need to consult the ASCII table in Appendix B.)

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

// Function prototypes
void fillArray(char [], int);
void showArray(const char [], int);

int main ()
{
    const int SIZE = 8;
    char prodCode[SIZE] = {'0', '0', '0', '0', '0', '0', '0', '0'};
    fillArray(prodCode, SIZE);
    showArray(prodCode, SIZE);
    return 0;
}

// Definition of function fillArray.
// (Hint: 65 is the ASCII code for 'A')
```

```
void fillArray(char arr[], int size)
{
   char code = 65;
   for (int k = 0; k < size; code++, k++)
       arr[k] = code;
}

// Definition of function showArray.

void showArray(const char codes[], int size)
{
   for (int k = 0; k < size; k++)
       cout << codes[k];
   cout << endl;
}
```

7.18 The following program skeleton, when completed, will ask the user to enter 10 integers, which are stored in an array. The function `avgArray`, which you must write, is to calculate and return the average of the numbers entered.

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

// Write your function prototype here

int main()
{
   const int SIZE = 10;
   int userNums[SIZE];

   cout << "Enter 10 numbers: ";
   for (int count = 0; count < SIZE; count++)
   {
       cout << "#" << (count + 1) << " ";
       cin >> userNums[count];
   }
   cout << "The average of those numbers is ";
   cout << avgArray(userNums, SIZE) << endl;
   return 0;
}

//
// Write the function avgArray here.
//
```

7.9 Two-Dimensional Arrays

CONCEPT: A two-dimensional array is like several identical arrays put together. It is useful for storing multiple sets of data.

An array is useful for storing and working with a set of data. Sometimes, though, it's necessary to work with multiple sets of data. For example, in a grade-averaging program a teacher might record all of one student's test scores in an array of `doubles`. If the teacher has 30 students, that means she'll need 30 arrays of `doubles` to record the scores for the entire class. Instead of defining 30 individual arrays, however, it would be better to define a two-dimensional array.

The arrays that you have studied so far are one-dimensional arrays. They are called *one-dimensional* because they can only hold one set of data. Two-dimensional arrays, which are sometimes called *2D arrays*, can hold multiple sets of data. It's best to think of a two-dimensional array as having rows and columns of elements, as shown in Figure 7-15. This figure shows an array of test scores, having three rows and four columns.

Figure 7-15

	Column 0	Column 1	Column 2	Column 3
Row 0	`scores[0] [0]`	`scores[0] [1]`	`scores[0] [2]`	`scores[0] [3]`
Row 1	`scores[1] [0]`	`scores[1] [1]`	`scores[1] [2]`	`scores[1] [3]`
Row 2	`scores[2] [0]`	`scores[2] [1]`	`scores[2] [2]`	`scores[2] [3]`

The array depicted in Figure 7-15 has three rows (numbered 0 through 2), and four columns (numbered 0 through 3). There are a total of 12 elements in the array.

To define a two-dimensional array, two size declarators are required. The first one is for the number of rows, and the second one is for the number of columns. Here is an example definition of a two-dimensional array with three rows and four columns:

```
double scores[3][4];
```

Rows   Columns

The first size declarator specifies the number of rows, and the second size declarator specifies the number of columns. Notice that each number is enclosed in its own set of brackets.

When processing the data in a two-dimensional array, each element has two subscripts: one for its row and another for its column. In the `scores` array defined above, the elements in row 0 are referenced as

```
scores[0][0]
scores[0][1]
scores[0][2]
scores[0][3]
```

The elements in row 1 are

```
scores[1][0]
scores[1][1]
scores[1][2]
scores[1][3]
```

And the elements in row 2 are

```
scores[2][0]
scores[2][1]
scores[2][2]
scores[2][3]
```

The subscripted references are used in a program just like the references to elements in a single-dimensional array, except now you use two subscripts. The first subscript represents the row position, and the second subscript represents the column position. For example, the following statement assigns the value 92.25 to the element at row 2, column 1 of the `scores` array:

```
scores[2][1] = 92.25;
```

And the following statement displays the element at row 0, column 2:

```
cout << scores[0][2];
```

Programs that cycle through each element of a two-dimensional array usually do so with nested loops. Program 7-21 is an example.

Program 7-21

```
// This program demonstrates a two-dimensional array.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    const int NUM_DIVS = 3;           // Number of divisions
    const int NUM_QTRS = 4;           // Number of quarters
    double sales[NUM_DIVS][NUM_QTRS]; // Array with 3 rows and 4 columns.
    double totalSales = 0;            // To hold the total sales.
    int div, qtr;                     // Loop counters.

    cout << "This program will calculate the total sales of\n";
    cout << "all the company's divisions.\n";
    cout << "Enter the following sales information:\n\n";

    // Nested loops to fill the array with quarterly
    // sales figures for each division.
    for (div = 0; div < NUM_DIVS; div++)
    {
        for (qtr = 0; qtr < NUM_QTRS; qtr++)
        {
            cout << "Division " << (div + 1);
            cout << ", Quarter " << (qtr + 1) << ": $";
            cin >> sales[div][qtr];
        }
        cout << endl; // Print blank line.
    }

    // Nested loops used to add all the elements.
    for (div = 0; div < NUM_DIVS; div++)
    {
        for (qtr = 0; qtr < NUM_QTRS; qtr++)
            totalSales += sales[div][qtr];
    }

    cout << fixed << showpoint << setprecision(2);
    cout << "The total sales for the company are: $";
    cout << totalSales << endl;
    return 0;
}
```

Program Output with Example Input Shown in Bold

```
This program will calculate the total sales of
all the company's divisions.
Enter the following sales data:

Division 1, Quarter 1: $31569.45 [Enter]
Division 1, Quarter 2: $29654.23 [Enter]
Division 1, Quarter 3: $32982.54 [Enter]
Division 1, Quarter 4: $39651.21 [Enter]

Division 2, Quarter 1: $56321.02 [Enter]
Division 2, Quarter 2: $54128.63 [Enter]
Division 2, Quarter 3: $41235.85 [Enter]
Division 2, Quarter 4: $54652.33 [Enter]

Division 3, Quarter 1: $29654.35 [Enter]
Division 3, Quarter 2: $28963.32 [Enter]
Division 3, Quarter 3: $25353.55 [Enter]
Division 3, Quarter 4: $32615.88 [Enter]

The total sales for the company are: $456782.34
```

When initializing a two-dimensional array, it helps to enclose each row's initialization list in a set of braces. Here is an example:

```
int hours[3][2] = {{8, 5}, {7, 9}, {6, 3}};
```

The same definition could also be written as:

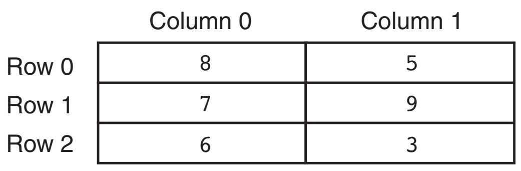
```
int hours[3][2] = {{8, 5},
                   {7, 9},
                   {6, 3}};
```

In either case, the values are assigned to hours in the following manner:

```
hours[0][0] is set to 8
hours[0][1] is set to 5
hours[1][0] is set to 7
hours[1][1] is set to 9
hours[2][0] is set to 6
hours[2][1] is set to 3
```

Figure 7-16 illustrates the initialization.

Figure 7-16



	Column 0	Column 1
Row 0	8	5
Row 1	7	9
Row 2	6	3

The extra braces that enclose each row's initialization list are optional. Both of the following statements perform the same initialization:

```
int hours[3][2] = {{8, 5}, {7, 9}, {6, 3}};
int hours[3][2] = {8, 5, 7, 9, 6, 3};
```

Because the extra braces visually separate each row, however, it's a good idea to use them. In addition, the braces give you the ability to leave out initializers within a row without omitting the initializers for the rows that follow it. For instance, look at the following array definition:

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

`table[0][0]` is initialized to 1, `table[1][0]` is initialized to 3, `table[1][1]` is initialized to 4, and `table[2][0]` is initialized to 5. `table[0][1]` and `table[2][1]` are not initialized. Because some of the array elements are initialized, these two initialized elements are automatically set to zero.

Passing Two-Dimensional Arrays to Functions

Program 7-22 demonstrates passing a two-dimensional array to a function. When a two-dimensional array is passed to a function, the parameter type must contain a size declarator for the number of columns. Here is the header for the function `showArray`, from Program 7-22:

```
void showArray(const int numbers[][COLS], int rows)
```

`COLS` is a global named constant which is set to 4. The function can accept any two-dimensional integer array, as long as it consists of four columns. In the program, the contents of two separate arrays are displayed by the function.

Program 7-22

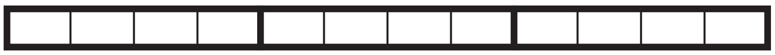
```
// This program demonstrates accepting a 2D array argument.
#include <iostream>
#include <iomanip>
using namespace std;

// Global constants
const int COLS = 4;       // Number of columns in each array
const int TBL1_ROWS = 3; // Number of rows in table1
const int TBL2_ROWS = 4; // Number of rows in table2

void showArray(const int [][COLS], int); // Function prototype

int main()
{
    int table1[TBL1_ROWS][COLS] = {{1, 2, 3, 4},
                                    {5, 6, 7, 8},
                                    {9, 10, 11, 12}};
    int table2[TBL2_ROWS][COLS] = {{10, 20, 30, 40},
                                    {50, 60, 70, 80},
                                    {90, 100, 110, 120},
                                    {130, 140, 150, 160}};

    cout << "The contents of table1 are:\n";
    showArray(table1, TBL1_ROWS);
    cout << "The contents of table2 are:\n";
    showArray(table2, TBL2_ROWS);
    return 0;
}

//*****************************************************************
// Function Definition for showArray                              *
// The first argument is a two-dimensional int array with COLS    *
// columns. The second argument, rows, specifies the number of    *
// rows in the array. The function displays the array's contents. *
//*****************************************************************

void showArray(const int numbers[][COLS], int rows)
{
    for (int x = 0; x < rows; x++)
    {
        for (int y = 0; y < COLS; y++)
        {
            cout << setw(4) << numbers[x][y] << " ";
        }
        cout << endl;
    }
}
```

Program Output

```
The contents of table1 are:
   1    2    3    4
   5    6    7    8
   9   10   11   12
The contents of table2 are:
  10   20   30   40
  50   60   70   80
  90  100  110  120
 130  140  150  160
```

C++ requires the columns to be specified in the function prototype and header because of the way two-dimensional arrays are stored in memory. One row follows another, as shown in Figure 7-17.

Figure 7-17



When the compiler generates code for accessing the elements of a two-dimensional array, it needs to know how many bytes separate the rows in memory. The number of columns is a critical factor in this calculation.

Summing All the Elements of a Two-Dimensional Array

To sum all the elements of a two-dimensional array, you can use a pair of nested loops to add the contents of each element to an accumulator. The following code is an example.

```
const int NUM_ROWS = 5;  // Number of rows
const int NUM_COLS = 5;  // Number of columns
int total = 0;           // Accumulator
int numbers[NUM_ROWS][NUM_COLS] = {{2, 7, 9, 6, 4},
                                    {6, 1, 8, 9, 4},
                                    {4, 3, 7, 2, 9},
                                    {9, 9, 0, 3, 1},
                                    {6, 2, 7, 4, 1}};

// Sum the array elements.
for (int row = 0; row < NUM_ROWS; row++)
{
    for (int col = 0; col < NUM_COLS; col++)
         total += numbers[row][col];
}

// Display the sum.
cout << "The total is " << total << endl;
```

Summing the Rows of a Two-Dimensional Array

Sometimes you may need to calculate the sum of each row in a two-dimensional array. For example, suppose a two-dimensional array is used to hold a set of test scores for a set of students. Each row in the array is a set of test scores for one student. To get the sum of a student's test scores (perhaps so an average may be calculated), you use a loop to add all the elements in one row. The following code shows an example.

```
const int NUM_STUDENTS = 3;      // Number of students
const int NUM_SCORES = 5;        // Number of test scores
double total;                    // Accumulator is set in the loops
double average;                  // To hold each student's average
double scores[NUM_STUDENTS][NUM_SCORES] = {{88, 97, 79, 86, 94},
                                            {86, 91, 78, 79, 84},
                                            {82, 73, 77, 82, 89}};

// Get each student's average score.
for (int row = 0; row < NUM_STUDENTS; row++)
{
    // Set the accumulator.
    total = 0;

    // Sum a row.
    for (int col = 0; col < NUM_SCORES; col++)
         total += scores[row][col];

    // Get the average.
    average = total / NUM_SCORES;

    // Display the average.
    cout << "Score average for student "
         << (row + 1) << " is " << average <<endl;
}
```

Notice that the `total` variable, which is used as an accumulator, is set to zero just before the inner loop executes. This is because the inner loop sums the elements of a row and stores the sum in `total`. Therefore, the `total` variable must be set to zero before each iteration of the inner loop.

Summing the Columns of a Two-Dimensional Array

Sometimes you may need to calculate the sum of each column in a two-dimensional array. In the previous example a two-dimensional array is used to hold a set of test scores for a set of students. Suppose you wish to calculate the class average for each of the test scores. To do this, you calculate the average of each column in the array. This is accomplished with a set of nested loops. The outer loop controls the column subscript and the inner loop controls the row subscript. The inner loop calculates the sum of a column, which is stored in an accumulator. The following code demonstrates.

```
const int NUM_STUDENTS = 3;    // Number of students
const int NUM_SCORES = 5;      // Number of test scores
double total;                  // Accumulator is set in the loops
double average;                // To hold each score's class average
double scores[NUM_STUDENTS][NUM_SCORES] = {{88, 97, 79, 86, 94},
                                           {86, 91, 78, 79, 84},
                                           {82, 73, 77, 82, 89}};

// Get the class average for each score.
for (int col = 0; col < NUM_SCORES; col++)
{
   // Reset the accumulator.
   total = 0;

   // Sum a column.
   for (int row = 0; row < NUM_STUDENTS; row++)
       total += scores[row][col];

   // Get the average.
   average = total / NUM_STUDENTS;

   // Display the class average.
   cout << "Class average for test " << (col + 1)
        << " is " << average << endl;
}
```

7.10 Arrays with Three or More Dimensions

CONCEPT: C++ does not limit the number of dimensions that an array may have. It is possible to create arrays with multiple dimensions, to model data that occur in multiple sets.

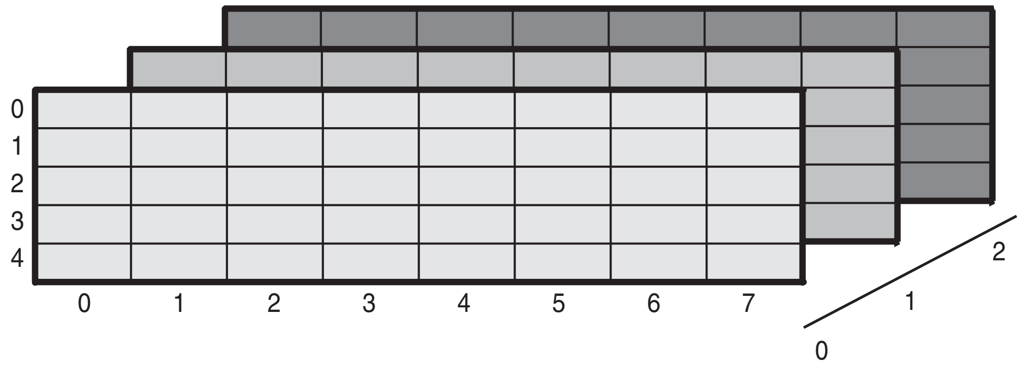
C++ allows you to create arrays with virtually any number of dimensions. Here is an example of a three-dimensional array definition:

```
double seats[3][5][8];
```

This array can be thought of as three sets of five rows, with each row containing eight elements. The array might be used to store the prices of seats in an auditorium, where there are eight seats in a row, five rows in a section, and a total of three sections.

Figure 7-18 illustrates the concept of a three-dimensional array as "pages" of two-dimensional arrays.

Figure 7-18



Arrays with more than three dimensions are difficult to visualize, but can be useful in some programming problems. For example, in a factory warehouse where cases of widgets are stacked on pallets, an array with four dimensions could be used to store a part number for each widget. The four subscripts of each element could represent the pallet number, case number, row number, and column number of each widget. Similarly, an array with five dimensions could be used if there were multiple warehouses.

NOTE: When writing functions that accept multi-dimensional arrays as arguments, all but the first dimension must be explicitly stated in the parameter list.

Checkpoint

7.19 Define a two-dimensional array of `int`s named `grades`. It should have 30 rows and 10 columns.

7.20 How many elements are in the following array?

```
double sales[6][4];
```

7.21 Write a statement that assigns the value 56893.12 to the first column of the first row of the array defined in Question 7.20.

7.22 Write a statement that displays the contents of the last column of the last row of the array defined in Question 7.20.

7.23 Define a two-dimensional array named `settings` large enough to hold the table of data below. Initialize the array with the values in the table.

12	24	32	21	42
14	67	87	65	90
19	1	24	12	8

7.24 Fill in the table below so it shows the contents of the following array:

```
int table[3][4] = {{2, 3}, {7, 9, 2}, {1}};
```


7.25 Write a function called `displayArray7`. The function should accept a two-dimensional array as an argument and display its contents on the screen. The function should work with any of the following arrays:

```
int hours[5][7];
int stamps[8][7];
int autos[12][7];
int cats[50][7];
```

7.26 A video rental store keeps DVDs on 50 racks with 10 shelves each. Each shelf holds 25 DVDs. Define a three-dimensional array large enough to represent the store's storage system.

7.11 Focus on Problem Solving and Program Design: A Case Study

The National Commerce Bank has hired you as a contract programmer. Your first assignment is to write a function that will be used by the bank's automated teller machines (ATMs) to validate a customer's personal identification number (PIN).

Your function will be incorporated into a larger program that asks the customer to input his or her PIN on the ATM's numeric keypad. (PINs are seven-digit numbers. The program stores each digit in an element of an integer array.) The program also retrieves a copy of the customer's actual PIN from a database. (The PINs are also stored in the database as seven-element arrays.) If these two numbers match, then the customer's identity is validated. Your function is to compare the two arrays and determine whether they contain the same numbers.

Here are the specifications your function must meet:

Parameters	The function is to accept as arguments two integer arrays of seven elements each. The first argument will contain the number entered by the customer. The second argument will contain the number retrieved from the bank's database.
Return value	The function should return a Boolean `true` value if the two arrays are identical. Otherwise, it should return `false`.

Here is the pseudocode for the function:

```
For each element in the first array
     Compare the element with the element in the second array
     that is in the corresponding position.
     If the two elements contain different values
        Return false.
     End If.
End For.
Return true.
```

The C++ code is shown below.

```
bool testPIN(const int custPIN[], const int databasePIN[], int size)
{
    for (int index = 0; index < size; index++)
    {
        if (custPIN[index] != databasePIN[index])
            return false; // We've found two different values.
    }
    return true; // If we make it this far, the values are the same.
}
```

Because you have only been asked to write a function that performs the comparison between the customer's input and the PIN that was retrieved from the database, you will also need to design a driver. Program 7-23 shows the complete program.

Program 7-23

```
// This program is a driver that tests a function comparing the
// contents of two int arrays.
#include <iostream>
using namespace std;

// Function Prototype
bool testPIN(const int [], const int [], int);

int main ()
{
    const int NUM_DIGITS = 7; // Number of digits in a PIN
    int pin1[NUM_DIGITS] = {2, 4, 1, 8, 7, 9, 0}; // Base set of values.
    int pin2[NUM_DIGITS] = {2, 4, 6, 8, 7, 9, 0}; // Only 1 element is
                                                   // different from pin1.
    int pin3[NUM_DIGITS] = {1, 2, 3, 4, 5, 6, 7}; // All elements are
                                                   // different from pin1.
    if (testPIN(pin1, pin2, NUM_DIGITS))
       cout << "ERROR: pin1 and pin2 report to be the same.\n";
    else
       cout << "SUCCESS: pin1 and pin2 are different.\n";

    if (testPIN(pin1, pin3, NUM_DIGITS))
       cout << "ERROR: pin1 and pin3 report to be the same.\n";
    else
       cout << "SUCCESS: pin1 and pin3 are different.\n";

    if (testPIN(pin1, pin1, NUM_DIGITS))
       cout << "SUCCESS: pin1 and pin1 report to be the same.\n";
    else
       cout << "ERROR: pin1 and pin1 report to be different.\n";
    return 0;
}

```

```
//*****************************************************************
// The following function accepts two int arrays. The arrays are  *
// compared. If they contain the same values, true is returned.   *
// If they contain different values, false is returned.           *
//*****************************************************************

bool testPIN(const int custPIN[], const int databasePIN[], int size)
{
    for (int index = 0; index < size; index++)
    {
        if (custPIN[index] != databasePIN[index])
           return false; // We've found two different values.
    }
    return true; // If we make it this far, the values are the same.
}
```

Program Output

```
SUCCESS: pin1 and pin2 are different.
SUCCESS: pin1 and pin3 are different.
SUCCESS: pin1 and pin1 report to be the same.
```

Case Study: See the Intersection of Sets Case Study on the book's companion Web site at www.pearsonhighered.com/gaddis.

7.12 If You Plan to Continue in Computer Science: Introduction to the STL `vector`

CONCEPT: The Standard Template Library offers a `vector` data type, which in many ways, is superior to standard arrays.

The *Standard Template Library* (STL) is a collection of data types and algorithms that you may use in your programs. These data types and algorithms are *programmer-defined*. They are not part of the C++ language, but were created in addition to the built-in data types. If you plan to continue your studies in the field of computer science, you should become familiar with the STL. This section introduces one of the STL data types. For more information on the STL, see Chapter 16.

The data types that are defined in the STL are commonly called *containers*. They are called containers because they store and organize data. There are two types of containers in the STL: sequence containers and associative containers. A *sequence container* organizes data in a sequential fashion, similar to an array. *Associative containers* organize data with keys, which allow rapid, random access to elements stored in the container.

In this section you will learn to use the `vector` data type, which is a sequence container. A `vector` is like an array in the following ways:

- A `vector` holds a sequence of values, or elements.
- A `vector` stores its elements in contiguous memory locations.
- You can use the array subscript operator `[]` to read the individual elements in the `vector`.

However, a `vector` offers several advantages over arrays. Here are just a few:

- You do not have to declare the number of elements that the `vector` will have.
- If you add a value to a `vector` that is already full, the `vector` will automatically increase its size to accommodate the new value.
- `vector`s can report the number of elements they contain.

Defining a `vector`

To use `vector`s in your program, you must include the `vector` header file with the following statement:

```
#include <vector>
```

NOTE: To use the `vector` data type, you must have the `using namespace std;` statement in your program.

Now you are ready to define an actual `vector` object. The syntax for defining a `vector` is somewhat different from the syntax used in defining a regular variable or array. Here is an example:

```
vector<int> numbers;
```

This statement defines `numbers` as a `vector` of `int`s. Notice that the data type is enclosed in angled brackets, immediately after the word `vector`. Because the `vector` expands in size as you add values to it, there is no need to declare a size. You can define a starting size, if you prefer. Here is an example:

```
vector<int> numbers(10);
```

This statement defines `numbers` as a `vector` of 10 `int`s. This is only a starting size, however. Although the `vector` has 10 elements, its size will expand if you add more than 10 values to it.

NOTE: If you specify a starting size for a `vector`, the size declarator is enclosed in parentheses, not square brackets.

When you specify a starting size for a `vector`, you may also specify an initialization value. The initialization value is copied to each element. Here is an example:

```
vector<int> numbers(10, 2);
```

In this statement, `numbers` is defined as a `vector` of 10 `int`s. Each element in `numbers` is initialized to the value 2.

You may also initialize a `vector` with the values in another `vector`. For example, look at the following statement. Assume that `set1` is a `vector` of `int`s that already has values stored in it.

```
vector<int> set2(set1);
```

After this statement executes, `set2` will be a copy of `set1`.

Table 7-3 summarizes the `vector` definition procedures we have discussed.

Table 7-3

Definition Format	Description
`vector<float> amounts;`	Defines `amounts` as an empty `vector` of `floats`.
`vector<string> names;`	Defines `names` as an empty `vector` of `string` objects.
`vector<int> scores(15);`	Defines `scores` as a `vector` of 15 `ints`.
`vector<char> letters(25, 'A');`	Defines `letters` as a `vector` of 25 characters. Each element is initialized with `'A'`.
`vector<double> values2(values1);`	Defines `values2` as a `vector` of `doubles`. All the elements of `values1`, which is also a `vector` of `doubles`, are copied to `value2`.

Using an Initialization List with a vector in C++ 11

11

If you are using C++ 11, you can initialize a `vector` with a list of values, as shown in this example:

```
vector<int> numbers { 10, 20, 30, 40 };
```

This statement defines a `vector` of `ints` named `numbers`. The `vector` will have 4 elements, initialized with the values 10, 20, 30, and 40. Notice that the initialization list is enclosed in a set of braces, but you do not use an = operator before the list.

Storing and Retrieving Values in a vector

To store a value in an element that already exists in a `vector`, you may use the array subscript operator `[]`. For example, look at Program 7-24.

Program 7-24

```
// This program stores, in two vectors, the hours worked by 5
// employees, and their hourly pay rates.
#include <iostream>
#include <iomanip>
#include <vector>       // Needed to define vectors
using namespace std;

int main()
{
    const int NUM_EMPLOYEES = 5;              // Number of employees
    vector<int> hours(NUM_EMPLOYEES);         // A vector of integers
    vector<double> payRate(NUM_EMPLOYEES);    // A vector of doubles
    int index;                                // Loop counter

    // Input the data.
    cout << "Enter the hours worked by " << NUM_EMPLOYEES;
    cout << " employees and their hourly rates.\n";
    for (index = 0; index < NUM_EMPLOYEES; index++)
```

(program continues)

Program 7-24 *(continued)*

```
19        {
20            cout << "Hours worked by employee #" << (index + 1);
21            cout << ": ";
22            cin >> hours[index];
23            cout << "Hourly pay rate for employee #";
24            cout << (index + 1) << ": ";
25            cin >> payRate[index];
26        }
27
28        // Display each employee's gross pay.
29        cout << "\nHere is the gross pay for each employee:\n";
30        cout << fixed << showpoint << setprecision(2);
31        for (index = 0; index < NUM_EMPLOYEES; index++)
32        {
33            double grossPay = hours[index] * payRate[index];
34            cout << "Employee #" << (index + 1);
35            cout << ": $" << grossPay << endl;
36        }
37        return 0;
38    }
```

Program Output with Example Input Shown in Bold

```
Enter the hours worked by 5 employees and their hourly rates.
Hours worked by employee #1: 10 [Enter]
Hourly pay rate for employee #1: 9.75 [Enter]
Hours worked by employee #2: 15 [Enter]
Hourly pay rate for employee #2: 8.62 [Enter]
Hours worked by employee #3: 20 [Enter]
Hourly pay rate for employee #3: 10.50 [Enter]
Hours worked by employee #4: 40 [Enter]
Hourly pay rate for employee #4: 18.75 [Enter]
Hours worked by employee #5: 40 [Enter]
Hourly pay rate for employee #5: 15.65 [Enter]

Here is the gross pay for each employee:
Employee #1: $97.50
Employee #2: $129.30
Employee #3: $210.00
Employee #4: $750.00
Employee #5: $626.00
```

Notice that Program 7-24 uses the following statements in lines 11 and 12 to define two vectors.

```
vector<int> hours(NUM_EMPLOYEES);       // A vector of integers
vector<double> payRate(NUM_EMPLOYEES); // A vector of doubles
```

Both of the vectors are defined with the starting size 5, which is the value of the named constant NUM_EMPLOYEES. The program uses the following loop in lines 18 through 26 to store a value in each element of both vectors:

```
for (index = 0; index < NUM_EMPLOYEES; index++)
{
   cout << "Hours worked by employee #" << (index + 1);
   cout << ": ";
   cin >> hours[index];
   cout << "Hourly pay rate for employee #";
   cout << (index + 1) << ": ";
   cin >> payRate[index];
}
```

Because the values entered by the user are being stored in vector elements that already exist, the program uses the array subscript operator [ ], as shown in the following statements, which appear in lines 22 and 25:

```
cin >> hours[index];

cin >> payRate[index];
```

Using the Range-Based for Loop with a vector in C++ 11

With C++ 11, you can use a range-based for loop to step through the elements of a vector, as shown in Program 7-25.

Program 7-25

```
// This program demonstrates the range-based for loop with a vector.
include <iostream>
#include <vector>
using namespace std;

int main()
{
    // Define and initialize a vector.
    vector<int> numbers { 10, 20, 30, 40, 50 };

    // Display the vector elements.
    for (int val : numbers)
        cout << val << endl;

    return 0;
}
```

Program Output

```
10
20
30
40
50
```

Program 7-26 shows how you can use a reference variable with the range-based `for` loop to store items in a `vector`.

Program 7-26

```
 1  // This program demonstrates the range-based for loop with a vector.
 2  #include <iostream>
 3  #include <vector>
 4  using namespace std;
 5
 6  int main()
 7  {
 8      // Define and initialize a vector.
 9      vector<int> numbers(5);
10
11      // Get values for the vector elements.
12      for (int &val : numbers)
13      {
14          cout << "Enter an integer value: ";
15          cin >> val;
16      }
17
18      // Display the vector elements.
19      cout << "Here are the values you entered:\n";
20      for (int val : numbers)
21          cout << val << endl;
22
23      return 0;
24  }
```

Program Output with Example Input Shown in Bold

```
Enter an integer value: 1 [Enter]
Enter an integer value: 2 [Enter]
Enter an integer value: 3 [Enter]
Enter an integer value: 4 [Enter]
Enter an integer value: 5 [Enter]
Here are the values you entered:
1
2
3
4
5
```

In line 9, we define `numbers` as a `vector` of `int`s, with 5 elements. Notice that in line 12 the range variable, `val`, has an ampersand (`&`) written in front of its name. This declares `val` as a reference variable. As the loop executes, the `val` variable will be an alias for a `vector` element. Any changes made to the `val` variable will actually be made to the `vector` element it references.

Also notice that in line 20 we did not declare `val` as a reference variable (there is no ampersand written in front of the variable's name). Because the loop is simply displaying the `vector` elements, and does not need to change the `vector`'s contents, there is no need to make `val` a reference variable.

Using the `push_back` Member Function

You cannot use the `[]` operator to access a `vector` element that does not exist. To store a value in a `vector` that does not have a starting size, or that is already full, use the `push_back` member function. The `push_back` member function accepts a value as an argument and stores that value after the last element in the `vector`. (It pushes the value onto the back of the `vector`.) Here is an example:

```
numbers.push_back(25);
```

Assuming `numbers` is a `vector` of `int`s, this statement stores 25 as the last element. If `numbers` is full, the statement creates a new last element and stores 25 in it. If there are no elements in `numbers`, this statement creates an element and stores 25 in it.

Program 7-27 is a modification of Program 7-24. This version, however, allows the user to specify the number of employees. The two `vector`s, `hours` and `payRate`, are defined without starting sizes. Because these `vector`s have no starting elements, the `push_back` member function is used to store values in the `vector`s.

Program 7-27

```
1  // This program stores, in two arrays, the hours worked by 5
2  // employees, and their hourly pay rates.
3  #include <iostream>
4  #include <iomanip>
5  #include <vector>    // Needed to define vectors
6  using namespace std;
7  
8  int main()
9  {
10     vector<int> hours;       // hours is an empty vector
11     vector<double> payRate;  // payRate is an empty vector
12     int numEmployees;        // The number of employees
13     int index;               // Loop counter
14 
15     // Get the number of employees.
16     cout << "How many employees do you have? ";
17     cin >> numEmployees;
18 
19     // Input the payroll data.
20     cout << "Enter the hours worked by " << numEmployees;
21     cout << " employees and their hourly rates.\n";
22     for (index = 0; index < numEmployees; index++)
23     {
24         int tempHours;           // To hold the number of hours entered
25         double tempRate;         // To hold the pay rate entered
26 
```

(program continues)

Program 7-27 *(continued)*

```
27            cout << "Hours worked by employee #" << (index + 1);
28            cout << ": ";
29            cin >> tempHours;
30            hours.push_back(tempHours);      // Add an element to hours
31            cout << "Hourly pay rate for employee #";
32            cout << (index + 1) << ": ";
33            cin >> tempRate;
34            payRate.push_back(tempRate); // Add an element to payRate
35        }
36
37        // Display each employee's gross pay.
38        cout << "Here is the gross pay for each employee:\n";
39        cout << fixed << showpoint << setprecision(2);
40        for (index = 0; index < numEmployees; index++)
41        {
42             double grossPay = hours[index] * payRate[index];
43             cout << "Employee #" << (index + 1);
44             cout << ": $" << grossPay << endl;
45        }
46        return 0;
47    }
```

Program Output with Example Input Shown in Bold

```
How many employees do you have? 3 [Enter]
Enter the hours worked by 3 employees and their hourly rates.
Hours worked by employee #1: 40 [Enter]
Hourly pay rate for employee #1: 12.63 [Enter]
Hours worked by employee #2: 25 [Enter]
Hourly pay rate for employee #2: 10.35 [Enter]
Hours worked by employee #3: 45 [Enter]
Hourly pay rate for employee #3: 22.65 [Enter]

Here is the gross pay for each employee:
Employee #1: $505.20
Employee #2: $258.75
Employee #3: $1019.2
```

Notice that in lines 40 through 45 the second loop, which calculates and displays each employee's gross pay, uses the `[]` operator to access the elements of the `hours` and `payRate` vectors:

```
for (index = 0; index < numEmployees; index++)
{
   double grossPay = hours[index] * payRate[index];
   cout << "Employee #" << (index + 1);
   cout << ": $" << grossPay << endl;
}
```

This is possible because the first loop in lines 22 through 35 uses the `push_back` member function to create the elements in the two vectors.

Determining the Size of a vector

Unlike arrays, vectors can report the number of elements they contain. This is accomplished with the size member function. Here is an example of a statement that uses the size member function:

```
numValues = set.size();
```

In this statement, assume that numValues is an int and set is a vector. After the statement executes, numValues will contain the number of elements in set.

The size member function is especially useful when you are writing functions that accept vectors as arguments. For example, look at the following code for the showValues function:

```
void showValues(vector<int> vect)
{
    for (int count = 0; count < vect.size(); count++)
        cout << vect[count] << endl;
}
```

Because the vector can report its size, this function does not need to accept a second argument indicating the number of elements in the vector. Program 7-28 demonstrates this function.

Program 7-28

```
// This program demonstrates the vector size
// member function.
#include <iostream>
#include <vector>
using namespace std;

// Function prototype
void showValues(vector<int>);

int main()
{
    vector<int> values;

    // Put a series of numbers in the vector.
    for (int count = 0; count < 7; count++)
        values.push_back(count * 2);

    // Display the numbers.
    showValues(values);
    return 0;
}

//**************************************************
// Definition of function showValues.              *
// This function accepts an int vector as its      *
// argument. The value of each of the vector's     *
// elements is displayed.                          *
//**************************************************
```

(program continues)

Program 7-28 *(continued)*

```

void showValues(vector<int> vect)
{
   for (int count = 0; count < vect.size(); count++)
      cout << vect[count] << endl;
}
```

Program Output

```
0
2
4
6
8
10
12
```

Removing Elements from a `vector`

Use the `pop_back` member function to remove the last element from a `vector`. In the following statement, assume that `collection` is the name of a `vector`.

```
collection.pop_back();
```

This statement removes the last element from the `collection` vector. Program 7-29 demonstrates the function.

Program 7-29

```
// This program demonstrates the vector pop_back member function.
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> values;

    // Store values in the vector.
    values.push_back(1);
    values.push_back(2);
    values.push_back(3);
    cout << "The size of values is " << values.size() << endl;

    // Remove a value from the vector.
    cout << "Popping a value from the vector...\n";
    values.pop_back();
    cout << "The size of values is now " << values.size() << endl;

```

```
        // Now remove another value from the vector.
        cout << "Popping a value from the vector...\n";
        values.pop_back();
        cout << "The size of values is now " << values.size() << endl;

        // Remove the last value from the vector.
        cout << "Popping a value from the vector...\n";
        values.pop_back();
        cout << "The size of values is now " << values.size() << endl;
        return 0;
}
```

Program Output

```
The size of values is 3
Popping a value from the vector...
The size of values is now 2
Popping a value from the vector...
The size of values is now 1
Popping a value from the vector...
The size of values is now 0
```

Clearing a vector

To completely clear the contents of a `vector`, use the `clear` member function, as shown in the following statement:

```
numbers.clear();
```

After this statement executes, `numbers` will be cleared of all its elements. Program 7-30 demonstrates the function.

Program 7-30

```
// This program demonstrates the vector clear member function.
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> values(100);

    cout << "The values vector has "
         << values.size() << " elements.\n";
    cout << "I will call the clear member function...\n";
    values.clear();
    cout << "Now, the values vector has "
         << values.size() << " elements.\n";
   return 0;
}
```

(program output continues)

Program 7-30 *(continued)*

Program Output

```
The values vector has 100 elements.
I will call the clear member function...
Now, the values vector has 0 elements.
```

Detecting an Empty `vector`

To determine if a `vector` is empty, use the `empty` member function. The function returns `true` if the `vector` is empty and `false` if the `vector` has elements stored in it. Assuming `numberVector` is a `vector`, here is an example of its use:

```
if (numberVector.empty())
   cout << "No values in numberVector.\n";
```

Program 7-31 uses a function named `avgVector`, which demonstrates the `empty` member function.

Program 7-31

```
// This program demonstrates the vector's empty member function.
#include <iostream>
#include <vector>
using namespace std;

// Function prototype
double avgVector(vector<int>);

int main()
{
    vector<int> values;   // A vector to hold values
    int numValues;        // The number of values
    double average;       // To hold the average

    // Get the number of values to average.
    cout << "How many values do you wish to average? ";
    cin >> numValues;

    // Get the values and store them in the vector.
    for (int count = 0; count < numValues; count++)
    {
        int tempValue;
        cout << "Enter a value: ";
        cin >> tempValue;
        values.push_back(tempValue);
    }

    // Get the average of the values and display it.
    average = avgVector(values);
    cout << "Average: " << average << endl;
    return 0;
}

//**************************************************************
// Definition of function avgVector.                           *
// This function accepts an int vector as its argument. If     *
// the vector contains values, the function returns the        *
// average of those values. Otherwise, an error message is     *
// displayed and the function returns 0.0.                     *
//**************************************************************

double avgVector(vector<int> vect)
{
    int total = 0;      // accumulator
    double avg;         // average

    if (vect.empty())   // Determine if the vector is empty
    {
        cout << "No values to average.\n";
        avg = 0.0;
    }
    else
    {
        for (int count = 0; count < vect.size(); count++)
            total += vect[count];
        avg = total / vect.size();
    }
    return avg;
}
```

Program Output with Example Input Shown in Bold

```
How many values do you wish to average? 5 [Enter]
Enter a value: 12
Enter a value: 18
Enter a value: 3
Enter a value: 7
Enter a value: 9
Average: 9
```

Program Output with Different Example Input Shown in Bold

```
How many values do you wish to average? 0 [Enter]
No values to average.
Average: 0
```

Summary of vector Member Functions

Table 7-4 provides a summary of the vector member function we have discussed, as well as some additional ones.

Table 7-4

Member Function	Description
`at(`*`element`*`)`	Returns the value of the element located at *element* in the `vector`. *Example:* `x = vect.at(5);` This statement assigns the value of the fifth element of `vect` to `x`.
`capacity()`	Returns the maximum number of elements that may be stored in the `vector` without additional memory being allocated. (This is not the same value as returned by the `size` member function). *Example:* `x = vect.capacity();` This statement assigns the capacity of `vect` to `x`.
`clear()`	Clears a `vector` of all its elements. *Example:* `vect.clear();` This statement removes all the elements from `vect`.

`empty()`	Returns true if the `vector` is empty. Otherwise, it returns false.
	Example:
	`if (vect.empty())`
	`cout << "The vector is empty.";`
	This statement displays the message if `vect` is empty.

`pop_back()`	Removes the last element from the `vector`. *Example:* `vect.pop_back();` This statement removes the last element of `vect`, thus reducing its size by 1.
`push_back(`*`value`*`)`	Stores a value in the last element of the `vector`. If the `vector` is full or empty, a new element is created. *Example:* `vect.push_back(7);` This statement stores 7 in the last element of `vect`.
`reverse()`	Reverses the order of the elements in the `vector`. (The last element becomes the first element, and the first element becomes the last element.) *Example:* `vect.reverse();` This statement reverses the order of the element in `vect`.

Table 7-4 *(continued)*

Member Function	Description
`resize(elements, value)`	Resizes a `vector` by *elements* elements. Each of the new elements is initialized with the value in *value*. *Example:* `vect.resize(5, 1);` This statement increases the size of `vect` by five elements. The five new elements are initialized to the value 1.
`swap(vector2)`	Swaps the contents of the `vector` with the contents of *vector2*. *Example:* `vect1.swap(vect2);` This statement swaps the contents of `vect1` and `vect2`

Checkpoint

7.27 What header file must you `#include` in order to define `vector` objects?

7.28 Write a definition statement for a `vector` named `frogs`. `frogs` should be an empty `vector` of `int`s.

7.29 Write a definition statement for a `vector` named `lizards`. `lizards` should be a `vector` of 20 `float`s.

7.30 Write a definition statement for a `vector` named `toads`. `toads` should be a `vector` of 100 `char`s, with each element initialized to `'Z'`.

7.31 `gators` is an empty `vector` of `int`s. Write a statement that stores the value 27 in `gators`.

7.32 `snakes` is a `vector` of `double`s, with 10 elements. Write a statement that stores the value 12.897 in element 4 of the `snakes vector`.

Review Questions and Exercises

Short Answer

1. What is the difference between a size declarator and a subscript?
2. Look at the following array definition.

```
int values[10];
```

How many elements does the array have?

What is the subscript of the first element in the array?

What is the subscript of the last element in the array?

Assuming that an int uses four bytes of memory, how much memory does the array use?

3. Why should a function that accepts an array as an argument, and processes that array, also accept an argument specifying the array's size?
4. Consider the following array definition:

   ```
   int values[5] = { 4, 7, 6, 8, 2 };
   ```

   What does each of the following statements display?

   ```
   cout << values[4] << endl; __________
   cout << (values[2] + values[3]) << endl; __________
   cout << ++values[1] << endl; __________
   ```

5. How do you define an array without providing a size declarator?
6. Look at the following array definition.

   ```
   int numbers[5] = { 1, 2, 3 };
   ```

   What value is stored in `numbers[2]`?

   What value is stored in `numbers[4]`?
7. Assuming that `array1` and `array2` are both arrays, why is it not possible to assign the contents of `array2` to `array1` with the following statement?

   ```
   array1 = array2;
   ```

8. Assuming that `numbers` is an array of `doubles`, will the following statement display the contents of the array?

   ```
   cout << numbers << endl;
   ```

9. Is an array passed to a function by value or by reference?
10. When you pass an array name as an argument to a function, what is actually being passed?
11. How do you establish a parallel relationship between two or more arrays?
12. Look at the following array definition.

    ```
    double sales[8][10];
    ```

    How many rows does the array have?

    How many columns does the array have?

    How many elements does the array have?

    Write a statement that stores a number in the last column of the last row in the array.
13. When writing a function that accepts a two-dimensional array as an argument, which size declarator must you provide in the parameter for the array?
14. What advantages does a `vector` offer over an array?

Fill-in-the-Blank

15. The _________ indicates the number of elements, or values, an array can hold.
16. The size declarator must be a(n) _________ with a value greater than _________.
17. Each element of an array is accessed and indexed by a number known as a(n) _________.
18. Subscript numbering in C++ always starts at _________.
19. The number inside the brackets of an array definition is the _________, but the number inside an array's brackets in an assignment statement, or any other statement that works with the contents of the array, is the _________.
20. C++ has no array _________ checking, which means you can inadvertently store data past the end of an array.

21. Starting values for an array may be specified with a(n) ________ list.
22. If an array is partially initialized, the uninitialized elements will be set to ________.
23. If the size declarator of an array definition is omitted, C++ counts the number of items in the ________ to determine how large the array should be.
24. By using the same ________ for multiple arrays, you can build relationships between the data stored in the arrays.
25. You cannot use the ________ operator to copy data from one array to another in a single statement.
26. Any time the name of an array is used without brackets and a subscript, it is seen as ________.
27. To pass an array to a function, pass the ________ of the array.
28. A(n) ________ array is like several arrays of the same type put together.
29. It's best to think of a two-dimensional array as having ________ and ________.
30. To define a two-dimensional array, ________ size declarators are required.
31. When initializing a two-dimensional array, it helps to enclose each row's initialization list in ________.
32. When a two-dimensional array is passed to a function the ________ size must be specified.
33. The ______________ is a collection of programmer-defined data types and algorithms that you may use in your programs.
34. The two types of containers defined by the STL are ________ and __________.
35. The `vector` data type is a(n) __________ container.
36. To define a `vector` in your program, you must `#include` the _________ header file.
37. To store a value in a `vector` that does not have a starting size, or that is already full, use the ___________ member function.
38. To determine the number of elements in a `vector`, use the _________ member function.
39. Use the ___________ member function to remove the last element from a `vector`.
40. To completely clear the contents of a `vector`, use the ________ member function.

Algorithm Workbench

41. `names` is an integer array with 20 elements. Write a regular `for` loop, as well as a range-based `for` loop that prints each element of the array.
42. The arrays `numberArray1` and `numberArray2` have 100 elements. Write code that copies the values in `numberArray1` to `numberArray2`.
43. In a program you need to store the identification numbers of 10 employees (as `int`s) and their weekly gross pay (as `double`s).
    A) Define two arrays that may be used in parallel to store the 10 employee identification numbers and gross pay amounts.
    B) Write a loop that uses these arrays to print each employee's identification number and weekly gross pay.

44. Define a two-dimensional array of integers named `grades`. It should have 30 rows and 10 columns.
45. In a program you need to store the populations of 12 countries.
    A) Define two arrays that may be used in parallel to store the names of the countries and their populations.
    B) Write a loop that uses these arrays to print each country's name and its population.
46. The following code totals the values in two arrays: `numberArray1` and `numberArray2`. Both arrays have 25 elements. Will the code print the correct sum of values for both arrays? Why or why not?

```
int total = 0;           // Accumulator
int count;               // Loop counter
// Calculate and display the total of the first array.
for (count = 0; count < 24; count++)
   total += numberArray1[count];
cout << "The total for numberArray1 is " << total << endl;
// Calculate and display the total of the second array.
for (count = 0; count < 24; count++)
   total += numberArray2[count];
cout << "The total for numberArray2 is " << total << endl;
```

47. Look at the following array definition.

```
int numberArray[9][11];
```

Write a statement that assigns 145 to the first column of the first row of this array.

Write a statement that assigns 18 to the last column of the last row of this array.

48. `values` is a two-dimensional array of `float`s with 10 rows and 20 columns. Write code that sums all the elements in the array and stores the sum in the variable `total`.
49. An application uses a two-dimensional array defined as follows.

```
int days[29][5];
```

Write code that sums each row in the array and displays the results.

Write code that sums each column in the array and displays the results.

True or False

50. T F An array's size declarator can be either a literal, a named constant, or a variable.
51. T F To calculate the amount of memory used by an array, multiply the number of elements by the number of bytes each element uses.
52. T F The individual elements of an array are accessed and indexed by unique numbers.
53. T F The first element in an array is accessed by the subscript 1.
54. T F The subscript of the last element in a single-dimensional array is one less than the total number of elements in the array.
55. T F The contents of an array element cannot be displayed with `cout`.
56. T F Subscript numbers may be stored in variables.
57. T F You can write programs that use invalid subscripts for an array.

58. T F Arrays cannot be initialized when they are defined. A loop or other means must be used.
59. T F The values in an initialization list are stored in the array in the order they appear in the list.
60. T F C++ allows you to partially initialize an array.
61. T F If an array is partially initialized, the uninitialized elements will contain "garbage."
62. T F If you leave an element uninitialized, you do not have to leave all the ones that follow it uninitialized.
63. T F If you leave out the size declarator of an array definition, you do not have to include an initialization list.
64. T F The uninitialized elements of a `string` array will automatically be set to the value `"0"`.
65. T F You cannot use the assignment operator to copy one array's contents to another in a single statement.
66. T F When an array name is used without brackets and a subscript, it is seen as the value of the first element in the array.
67. T F To pass an array to a function, pass the name of the array.
68. T F When defining a parameter variable to hold a single-dimensional array argument, you do not have to include the size declarator.
69. T F When an array is passed to a function, the function has access to the original array.
70. T F A two-dimensional array is like several identical arrays put together.
71. T F It's best to think of two-dimensional arrays as having rows and columns.
72. T F The first size declarator (in the declaration of a two-dimensional array) represents the number of columns. The second size definition represents the number of rows.
73. T F Two-dimensional arrays may be passed to functions, but the row size must be specified in the definition of the parameter variable.
74. T F C++ allows you to create arrays with three or more dimensions.
75. T F A `vector` is an associative container.
76. T F To use a `vector`, you must include the `vector` header file.
77. T F `vector`s can report the number of elements they contain.
78. T F You can use the `[]` operator to insert a value into a `vector` that has no elements.
79. T F If you add a value to a `vector` that is already full, the `vector` will automatically increase its size to accommodate the new value.

Find the Error

Each of the following definitions and program segments has errors. Locate as many as you can.

80.
```
int size;
double values[size];
```

81.
```
int collection[-20];
```

82.
```
int table[10];
for (int x = 0; x < 20; x++)
{
   cout << "Enter the next value: ";
   cin >> table[x];
}
```

83. `int hours[3] = 8, 12, 16;`

84. `int numbers[8] = {1, 2, , 4, , 5};`

85. `float ratings[];`

86.
```
char greeting[] = {'H', 'e', 'l', 'l', 'o'};
cout << greeting;
```

87.
```
int array1[4], array2[4] = {3, 6, 9, 12};
array1 = array2;
```

88.
```
void showValues(int nums)
{
   for (int count = 0; count < 8; count++)
      cout << nums[count];
}
```

89.
```
void showValues(int nums[4][])
{
   for (rows = 0; rows < 4; rows++)
      for (cols = 0; cols < 5; cols++)
         cout << nums[rows][cols];
}
```

90. `vector<int> numbers = { 1, 2, 3, 4 };`

Programming Challenges

1. Largest/Smallest Array Values

Write a program that lets the user enter 10 values into an array. The program should then display the largest and smallest values stored in the array.

2. Rainfall Statistics

Write a program that lets the user enter the total rainfall for each of 12 months into an array of `doubles`. The program should calculate and display the total rainfall for the year, the average monthly rainfall, and the months with the highest and lowest amounts.

Input Validation: Do not accept negative numbers for monthly rainfall figures.

3. Chips and Salsa

VideoNote
Solving the Chips and Salsa Problem

Write a program that lets a maker of chips and salsa keep track of sales for five different types of salsa: mild, medium, sweet, hot, and zesty. The program should use two parallel 5-element arrays: an array of strings that holds the five salsa names and an array of integers that holds the number of jars sold during the past month for each salsa type. The salsa names should be stored using an initialization list at the time the name array is created. The program should prompt the user to enter the number of jars sold for each type. Once this sales data has been entered, the program should produce a report that displays sales for each salsa type, total sales, and the names of the highest selling and lowest selling products.

Input Validation: Do not accept negative values for number of jars sold.

4. **Larger Than *n***

In a program, write a function that accepts three arguments: an array, the size of the array, and a number *n*. Assume that the array contains integers. The function should display all of the numbers in the array that are greater than the number *n*.

5. **Monkey Business**

A local zoo wants to keep track of how many pounds of food each of its three monkeys eats each day during a typical week. Write a program that stores this information in a two-dimensional 3 × 5 array, where each row represents a different monkey and each column represents a different day of the week. The program should first have the user input the data for each monkey. Then it should create a report that includes the following information:

- Average amount of food eaten per day by the whole family of monkeys.
- The least amount of food eaten during the week by any one monkey.
- The greatest amount of food eaten during the week by any one monkey.

Input Validation: Do not accept negative numbers for pounds of food eaten.

6. **Rain or Shine**

An amateur meteorologist wants to keep track of weather conditions during the past year's three-month summer season and has designated each day as either rainy ('R'), cloudy ('C'), or sunny ('S'). Write a program that stores this information in a 3 × 30 array of characters, where the row indicates the month (0 = June, 1 = July, 2 = August) and the column indicates the day of the month. Note that data are not being collected for the 31st of any month. The program should begin by reading the weather data in from a file. Then it should create a report that displays, for each month and for the whole three-month period, how many days were rainy, how many were cloudy, and how many were sunny. It should also report which of the three months had the largest number of rainy days. Data for the program can be found in the `RainOrShine.txt` file.

7. **Number Analysis Program**

Write a program that asks the user for a file name. Assume the file contains a series of numbers, each written on a separate line. The program should read the contents of the file into an array and then display the following data:

- The lowest number in the array
- The highest number in the array
- The total of the numbers in the array
- The average of the numbers in the array

If you have downloaded this book's source code from the companion Web site, you will find a file named numbers.txt in the Chapter 07 folder. You can use the file to test the program. (The companion Web site is at www.pearsonhighered.com/gaddis.)

8. **Lo Shu Magic Square**

The Lo Shu Magic Square is a grid with 3 rows and 3 columns shown in Figure 7-19. The Lo Shu Magic Square has the following properties:

- The grid contains the numbers 1 through 9 exactly.
- The sum of each row, each column, and each diagonal all add up to the same number. This is shown in Figure 7-20.

In a program you can simulate a magic square using a two-dimensional array. Write a function that accepts a two-dimensional array as an argument, and determines whether the array is a Lo Shu Magic Square. Test the function in a program.

Figure 7-19

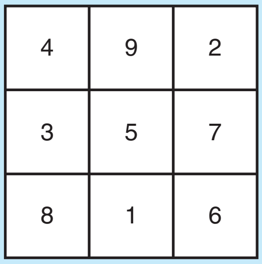
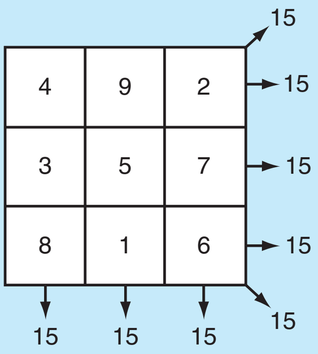


Figure 7-20



9. **Payroll**

Write a program that uses the following arrays:

- `empId`: an array of seven long integers to hold employee identification numbers. The array should be initialized with the following numbers:

```
5658845   4520125   7895122   8777541
8451277   1302850   7580489
```

- `hours`: an array of seven integers to hold the number of hours worked by each employee
- `payRate`: an array of seven `doubles` to hold each employee's hourly pay rate
- `wages`: an array of seven `doubles` to hold each employee's gross wages

The program should relate the data in each array through the subscripts. For example, the number in element 0 of the `hours` array should be the number of hours worked by the employee whose identification number is stored in element 0 of the `empId` array. That same employee's pay rate should be stored in element 0 of the `payRate` array.

The program should display each employee number and ask the user to enter that employee's hours and pay rate. It should then calculate the gross wages for that employee (hours times pay rate) and store them in the `wages` array. After the data has been entered for all the employees, the program should display each employee's identification number and gross wages.

Input Validation: Do not accept negative values for hours or numbers less than 15.00 for pay rate.

10. Driver's License Exam

The local Driver's License Office has asked you to write a program that grades the written portion of the driver's license exam. The exam has 20 multiple choice questions. Here are the correct answers:

1. A	6. B	11. A	16. C
2. D	7. A	12. C	17. C
3. B	8. B	13. D	18. A
4. B	9. C	14. B	19. D
5. C	10. D	15. D	20. B

Your program should store the correct answers shown above in an array. It should ask the user to enter the student's answers for each of the 20 questions, and the answers should be stored in another array. After the student's answers have been entered, the program should display a message indicating whether the student passed or failed the exam. (A student must correctly answer 15 of the 20 questions to pass the exam.) It should then display the total number of correctly answered questions, the total number of incorrectly answered questions, and a list showing the question numbers of the incorrectly answered questions.

Input Validation: Only accept the letters A, B, C, or D as answers.

11. Exam Grader

One of your professors has asked you to write a program to grade her final exams, which consist of only 20 multiple-choice questions. Each question has one of four possible answers: A, B, C, or D. The file CorrectAnswers.txt contains the correct answers for all of the questions, with each answer written on a separate line. The first line contains the answer to the first question, the second line contains the answer to the second question, and so forth. (Download the book's source code from the companion Web site at www.pearsonhighered.com/gaddis. You will find the file in the Chapter 07 folder.)

Write a program that reads the contents of the CorrectAnswers.txt file into a `char` array, and then reads the contents of another file, containing a student's answers, into a second `char` array. (You can use the file StudentAnswers.txt for testing purposes. This file is also in the Chapter 07 source code folder, available on the book's companion Web site.) The program should determine the number of questions that the student missed and then display the following:

- A list of the questions missed by the student, showing the correct answer and the incorrect answer provided by the student for each missed question

- The total number of questions missed
- The percentage of questions answered correctly. This can be calculated as

  Correctly Answered Questions ÷ Total Number of Questions

- If the percentage of correctly answered questions is 70% or greater, the program should indicate that the student passed the exam. Otherwise, it should indicate that the student failed the exam.

12. Grade Book

A teacher has five students who have taken four tests. The teacher uses the following grading scale to assign a letter grade to a student, based on the average of his or her four test scores.

Test Score	Letter Grade
90–100	A
80–89	B
70–79	C
60–69	D
0–59	F

Write a program that uses an array of `string` objects to hold the five student names, an array of five characters to hold the five students' letter grades, and five arrays of four `doubles` to hold each student's set of test scores.

The program should allow the user to enter each student's name and his or her four test scores. It should then calculate and display each student's average test score and a letter grade based on the average.

Input Validation: Do not accept test scores less than 0 or greater than 100.

13. Grade Book Modification

Modify the grade book application in Programming Challenge 13 so it drops each student's lowest score when determining the test score averages and letter grades.

14. Lottery Application

Write a program that simulates a lottery. The program should have an array of five integers named `lottery` and should generate a random number in the range of 0 through 9 for each element in the array. The user should enter five digits, which should be stored in an integer array named `user`. The program is to compare the corresponding elements in the two arrays and keep a count of the digits that match. For example, the following shows the `lottery` array and the `user` array with sample numbers stored in each. There are two matching digits (elements 2 and 4).

lottery array:

7	4	9	1	3

user array:

4	2	9	7	3

The program should display the random numbers stored in the `lottery` array and the number of matching digits. If all of the digits match, display a message proclaiming the user as a grand prize winner.

15. `vector` Modification

Modify the National Commerce Bank case study presented in Program 7-23 so `pin1`, `pin2`, and `pin3` are `vectors` instead of arrays. You must also modify the `testPIN` function to accept a `vector` instead of an array.

16. World Series Champions

If you have downloaded this book's source code from the companion Web site, you will find the following files in this chapter's folder:

- Teams.txt—This file contains a list of several Major League baseball teams in alphabetical order. Each team listed in the file has won the World Series at least once.
- WorldSeriesWinners.txt—This file contains a chronological list of the World Series' winning teams from 1903 to 2012. (The first line in the file is the name of the team that won in 1903, and the last line is the name of the team that won in 2012. Note that the World Series was not played in 1904 or 1994.)

Write a program that displays the contents of the Teams.txt file on the screen and prompts the user to enter the name of one of the teams. The program should then display the number of times that team has won the World Series in the time period from 1903 to 2012.

TIP: Read the contents of the WorldSeriesWinners.txt file into an array or `vector`. When the user enters the name of a team, the program should step through the array or `vector` counting the number of times the selected team appears.

17. Name Search

If you have downloaded this book's source code from the companion Web site, you will find the following files in this chapter's folder:

- GirlNames.txt—This file contains a list of the 200 most popular names given to girls born in the United States from 2000 to 2009.
- BoyNames.txt—This file contains a list of the 200 most popular names given to boys born in the United States from 2000 to 2009.

Write a program that reads the contents of the two files into two separate arrays or `vectors`. The user should be able to enter a boy's name, a girl's name, or both, and the application should display messages indicating whether the names were among the most popular.

18. Tic-Tac-Toe Game

Write a program that allows two players to play a game of tic-tac-toe. Use a two-dimensional `char` array with three rows and three columns as the game board. Each element of the array should be initialized with an asterisk (*). The program should run a loop that

- Displays the contents of the board array
- Allows player 1 to select a location on the board for an X. The program should ask the user to enter the row and column number.

- Allows player 2 to select a location on the board for an O. The program should ask the user to enter the row and column number.
- Determines whether a player has won, or a tie has occurred. If a player has won, the program should declare that player the winner and end. If a tie has occurred, the program should say so and end.

Player 1 wins when there are three Xs in a row on the game board. The Xs can appear in a row, in a column, or diagonally across the board. A tie occurs when all of the locations on the board are full, but there is no winner.

19. 2D Array Operations

Write a program that creates a two-dimensional array initialized with test data. Use any data type you wish. The program should have the following functions:

- **`getTotal`**. This function should accept a two-dimensional array as its argument and return the total of all the values in the array.
- **`getAverage`**. This function should accept a two-dimensional array as its argument and return the average of all the values in the array.
- **`getRowTotal`**. This function should accept a two-dimensional array as its first argument and an integer as its second argument. The second argument should be the subscript of a row in the array. The function should return the total of the values in the specified row.
- **`getColumnTotal`**. This function should accept a two-dimensional array as its first argument and an integer as its second argument. The second argument should be the subscript of a column in the array. The function should return the total of the values in the specified column.
- **`getHighestInRow`**. This function should accept a two-dimensional array as its first argument and an integer as its second argument. The second argument should be the subscript of a row in the array. The function should return the highest value in the specified row of the array.
- **`getLowestInRow`**. This function should accept a two-dimensional array as its first argument and an integer as its second argument. The second argument should be the subscript of a row in the array. The function should return the lowest value in the specified row of the array.

Demonstrate each of the functions in this program.

Group Project

20. Theater Seating

This program should be designed and written by a team of students. Here are some suggestions:

- One student should design function `main`, which will call the other functions in the program. The remainder of the functions will be designed by other members of the team.
- The requirements of the program should be analyzed so each student is given about the same work load.
- The parameters and return types of each function should be decided in advance.
- The program can be implemented as a multi-file program, or all the functions can be cut and pasted into the main file.

Here is the assignment: Write a program that can be used by a small theater to sell tickets for performances. The theater's auditorium has 15 rows of seats, with 30 seats in each row. The program should display a screen that shows which seats are available and which are taken. For example, the following screen shows a chart depicting each seat in the theater. Seats that are taken are represented by an * symbol, and seats that are available are represented by a # symbol:

```
                       Seats
           123456789012345678901234567890
Row  1     ***###***###*########*****####
Row  2     ####*************####*******##
Row  3     **###**********########****###
Row  4     **######**************##******
Row  5     ********#####*********########
Row  6     ##############************####
Row  7     #######************###########
Row  8     ************##****############
Row  9     #########*****############****
Row  10    #####*************############
Row  11    #**********#################**
Row  12    #############********########*
Row  13    ###***********########**######
Row  14    ##############################
Row  15    ##############################
```

Here is a list of tasks this program must perform:

- When the program begins, it should ask the user to enter the seat prices for each row. The prices can be stored in a separate array. (Alternatively, the prices may be read from a file.)
- Once the prices are entered, the program should display a seating chart similar to the one shown above. The user may enter the row and seat numbers for tickets being sold. Every time a ticket or group of tickets is purchased, the program should display the total ticket prices and update the seating chart.
- The program should keep a total of all ticket sales. The user should be given an option of viewing this amount.
- The program should also give the user an option to see a list of how many seats have been sold, how many seats are available in each row, and how many seats are available in the entire auditorium.

Input Validation: When tickets are being sold, do not accept row or seat numbers that do not exist. When someone requests a particular seat, the program should make sure that seat is available before it is sold.