Chapter 7
Arrays and Vectors

Topics

7.1 Arrays Hold Multiple Values

7.2 Accessing Array Elements

7.3 No Bounds Checking in C++

7.4 The Range-Based `for` Loop

7.5 Processing Array Contents

7.6 Focus on Software Engineering: Using Parallel Arrays

7.7 Arrays as Function Arguments

7.8 Two-Dimensional Arrays

7.9 Arrays with Three or More Dimensions

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

7.11 Introduction to the STL `vector`

Printed by: cba92c7af985df9@placeholder.78751.edu. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Chapter 7
Arrays and Vectors

Topics

7.1 Arrays Hold Multiple Values

7.2 Accessing Array Elements

7.3 No Bounds Checking in C++

7.4 The Range-Based `for` Loop

7.5 Processing Array Contents

7.6 Focus on Software Engineering: Using Parallel Arrays

7.7 Arrays as Function Arguments

7.8 Two-Dimensional Arrays

7.9 Arrays with Three or More Dimensions

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

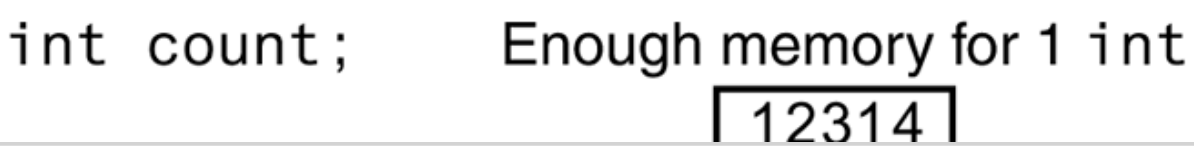
7.11 Introduction to the STL `vector`

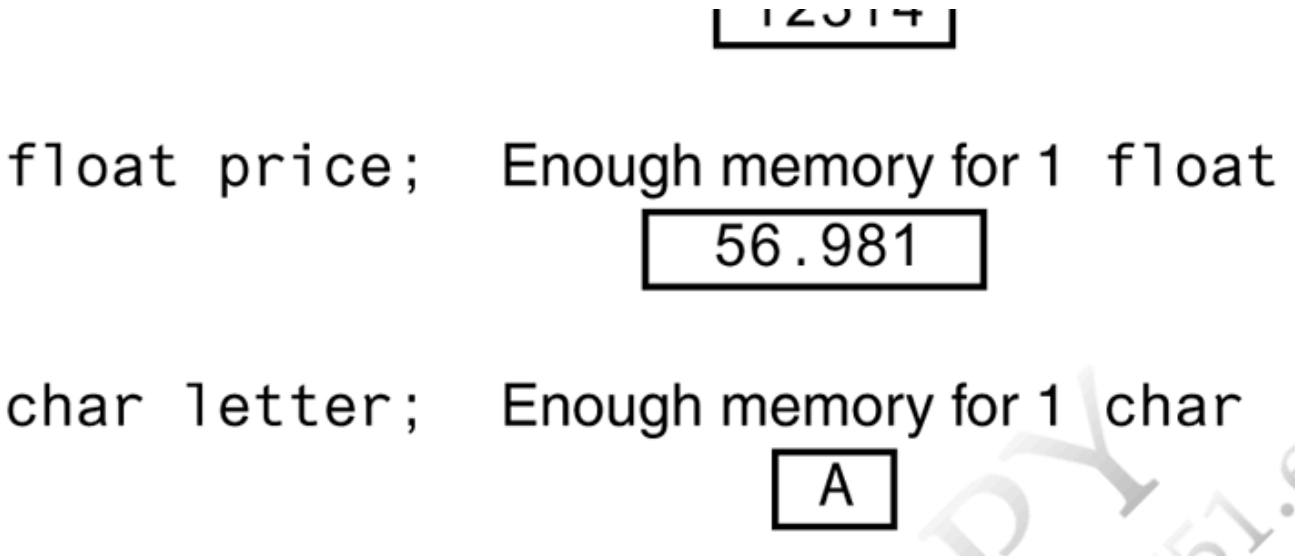
7.1 Arrays Hold Multiple Values

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 Variables can hold one value at a time



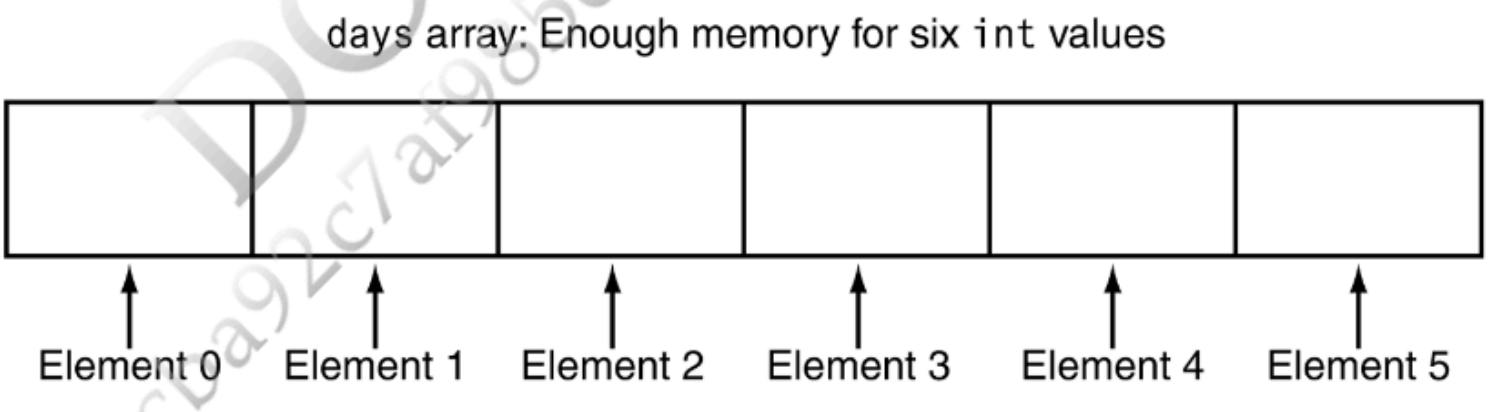


An array works like a variable that can store a group of values, all of the same type. The values are stored together in consecutive memory locations. Here is a definition of an array of integers:

```
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 with six elements



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 system, a `short` uses 2 bytes of memory, so the `hours` array would occupy 12 bytes. This is shown in Figure 7-3.

Figure 7-3 Array memory usage

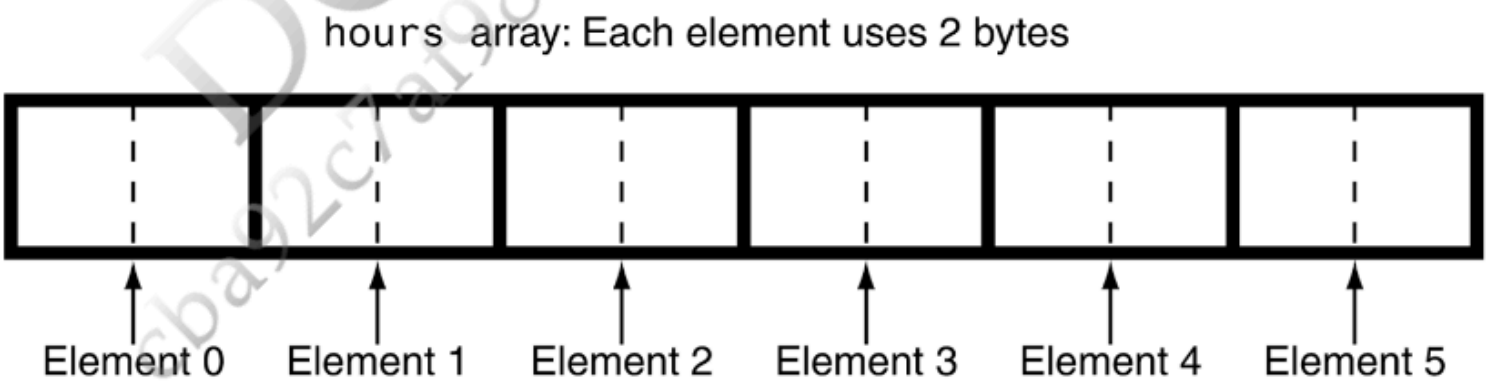


Figure 7-3 Full Alternative Text

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 Arrays and Their sizes

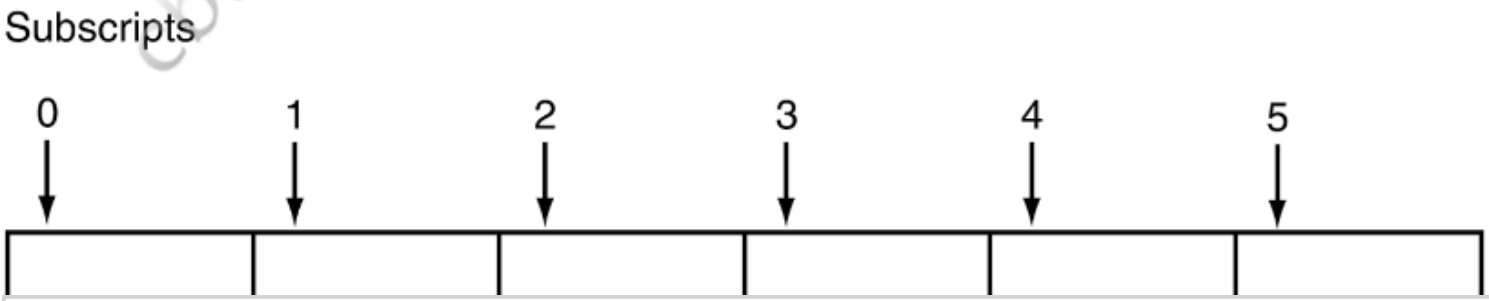
Array Definition	Number of Elements	Size of Each Element	Size of the Array
`char letters[25];`	25	1 byte	25 bytes
`short rings[100];`	100	2 bytes	200 bytes
`int miles[84];`	84	4 bytes	336 bytes
`float temp[12];`	12	4 bytes	48 bytes
`double distance[1000];`	1000	8 bytes	8000 bytes

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 Subscripts



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:

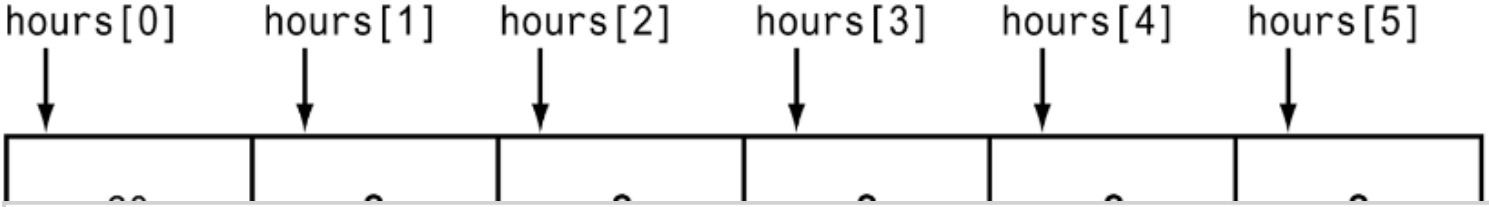
```
hours[0] = 20;
```

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 `hours` array after the statement assigns 20 to `hours[0]`.

Figure 7-5 Contents of the hours array



20	?	?	?	?	?

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 Contents of the hours array

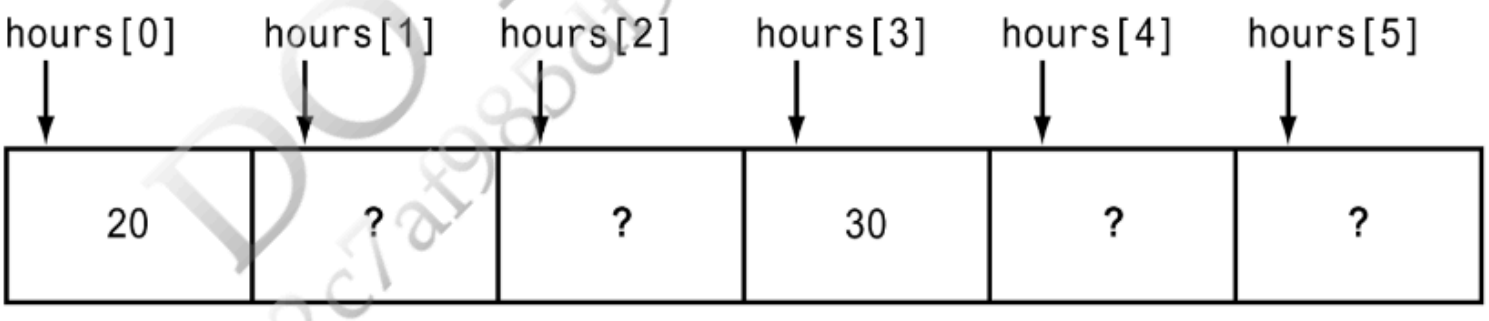


Figure 7-6 Full Alternative Text

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

```
 1  // This program asks for the number of hours worked
 2  // by six employees. It stores the values in an array.
 3  #include
 4  using namespace std;
 5
 6  int main()
 7  {
 8      const int NUM_EMPLOYEES = 6;
 9      int hours[NUM_EMPLOYEES];
10
11      // Get the hours worked by each employee.
12      cout << "Enter the hours worked by "
13           << NUM_EMPLOYEES << " employees: ";
14      cin >> hours[0];
15      cin >> hours[1];
16      cin >> hours[2];
17      cin >> hours[3];
18      cin >> hours[4];
19      cin >> hours[5];
20
21      // Display the values in the array.
22      cout << "The hours you entered are:";
23      cout << " " << hours[0];
24      cout << " " << hours[1];
25      cout << " " << hours[2];
26      cout << " " << hours[3];
27      cout << " " << hours[4];
28      cout << " " << hours[5] << endl;
29      return 0;
30  }
```

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
```

The `while` loop in lines 17 and 18 reads items from the file and assigns them to elements of the `numbers` array. Notice 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 7-8 demonstrates this.

Program 7-8

```
1  // This program writes the contents of an array to a file.
2  #include
3  #include
4  using namespace std;
5
6  int main()
7  {
8      const int ARRAY_SIZE = 10;  // Array size
9      int numbers[ARRAY_SIZE];    // Array with 10 elements
10     int count;                  // Loop counter variable
11     ofstream outputFile;        // Output file stream object
12
13     // Store values in the array.
14     for (count = 0; count < ARRAY_SIZE; count++)
15        numbers[count] = count;
16
17     // Open a file for output.
18     outputFile.open("SavedNumbers.txt");
19
20     // Write the array contents to the file.
21     for (count = 0; count < ARRAY_SIZE; count++)
22        outputFile << numbers[count] << endl;
23
24     // Close the file.
25     outputFile.close();
26
```

```
26
27      // That's it!
28      cout << "The numbers were saved to the file.\n ";
29      return 0;
30  }
```

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 7-9 demonstrates this capability.

Warning!

Program 7-9 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 7-9

```
1  // This program unsafely accesses an area of memory by writing
2  // values beyond an array's boundary.
3  // WARNING: If you compile and run this program, it could crash.
4  #include
5  using namespace std;
6
7  int main()
8  {
9      const int SIZE = 3;  // Constant for the array size
10     int values[SIZE];    // An array of 3 integers
11     int count;           // Loop counter variable
12
13     // Attempt to store five numbers in the 3-element array.
14     cout << "I will store 5 numbers in a 3-element array!\n";
15     for (count = 0; count < 5; count++)
16         values[count] = 100;
17
18     // If the program is still running, display the numbers.
19     cout << "If you see this message, it means the program\n";
20     cout << "has not crashed! Here are the numbers:\n";
21     for (count = 0; count < 5; count++)
22         cout << values[count] << endl;
```

```
22          cout << values[count] << endl;
23      return 0;
24  }
```

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 7-11 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 7-11 Writing data beyond the bounds of an array

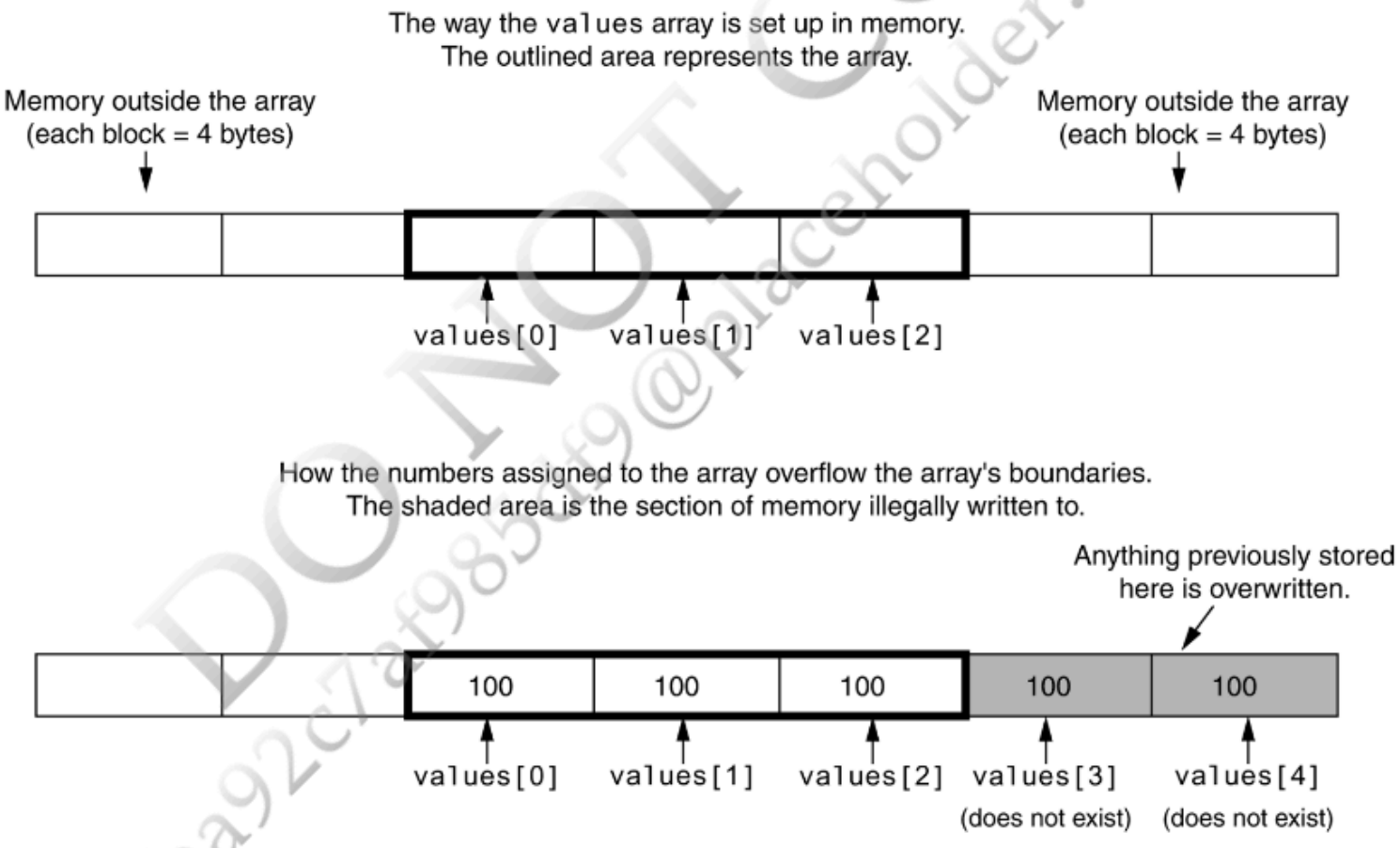


Figure 7-11 Full Alternative Text

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.

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 4-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
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;
}
```

7.8 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.)

A. `int numbers[10] = {0, 0, 1, 0, 0, 1, 0, 0, 1, 1};`

B. `int matrix[5] = {1, 2, 3, 4, 5, 6, 7};`

C. `double radii[10] = {3.2, 4.7};`

D. `int table[7] = {2, , , 27, , 45, 39};`

E. `char codes[] = {'A', 'X', '1', '2', 's'};`

F. `int blanks[];`

7.8 Define the following arrays:

A. `ages`, a 10-element array of `ints` initialized with the values 5, 7, 9, 14, 15, 17, 18, 19, 21, and 23.
B. `temps`, a 7-element array of `floats` 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 `chars` 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.)

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

7.4 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 introduced 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 8-element array, the loop will iterate 8 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 to which the elements can automatically be converted.
- *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
```

```
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

```
1  // This program demonstrates the range-based for loop.
2  #include
3  using namespace std;
4
5  int main()
6  {
7      // Define an array of integers.
```

```
 7      // Define an array of integers.
 8      int numbers[] = { 10, 20, 30, 40, 50 };
 9
10      // Display the values in the array.
11      for (int val : numbers)
12          cout << val << endl;
13
14      return 0;
15  }
```

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

```
 1  // This program demonstrates the range-based for loop.
 2  #include
 3  #include
 4  using namespace std;
 5
 6  int main()
 7  {
 8      string planets[] = { "Mercury", "Venus", "Earth", "Mars",
 9                           "Jupiter", "Saturn", "Uranus",
10                           "Neptune", "Pluto (a dwarf planet)" };
11
12      cout << "Here are the planets:\n";
13
14      // Display the values in the array.
15      for (string val : planets)
```

```
16          cout << val << endl;
17
18      return 0;
19  }
```

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

Also notice 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;
}
```

7.5 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

```
1  // This program stores, in an array, the hours worked by
2  // employees who all make the same hourly wage.
3  #include
```

```
 4  #include
 5  using namespace std;
 6
 7  int main()
 8  {
 9      const int NUM_EMPLOYEES = 5; // Number of employees
10      int hours[NUM_EMPLOYEES];    // Array to hold hours
11      double payrate;              // Hourly pay rate
12      double grossPay;             // To hold the gross pay
13
14      // Input the hours worked.
15      cout << "Enter the hours worked by ";
16      cout << NUM_EMPLOYEES << " employees who all\n";
17      cout << "earn the same hourly rate.\n";
18      for (int index = 0; index < NUM_EMPLOYEES; index++)
19      {
20          cout << "Employee #" << (index + 1) << ": ";
21          cin >> hours[index];
22      }
23
24      // Input the hourly rate for all employees.
25      cout << "Enter the hourly pay rate for all the employees:  ";
26      cin >> payrate;
27
28      // Display each employee's gross pay.
29      cout << "Here is the gross pay for each employee:\n";
30      cout << fixed << showpoint << setprecision(2);
31      for (int index = 0; index < NUM_EMPLOYEES; index++)
32      {
33          grossPay = hours[index] * payrate;
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 who all
earn the same hourly rate.
Employee #1: 5 Enter
Employee #2: 10 Enter
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:
```

```
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`:

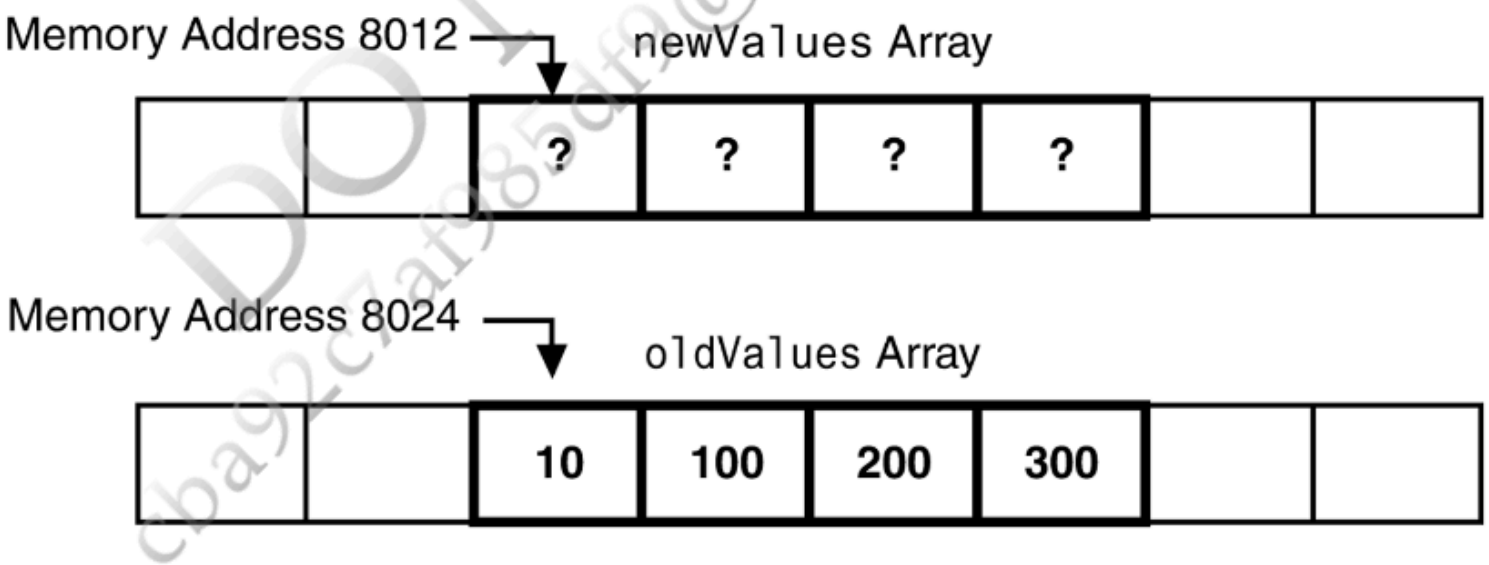
```
newValues = oldValues; // Wrong!
```

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 The `newValues` and `oldValues` arrays



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 **Array Expressions**

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 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 the `numbers` array.

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

variables that are listed inside the square brackets must be the same as the number of elements in the array. Otherwise, a compiler error will occur.

Let's look at an example. Suppose we have three values stored in an array, as shown here:

```
int testScores[] = {80, 90, 100};
```

The following statement is a structured binding declaration that unpacks the array and assigns its values to variables:

```
auto [score1, score2, score3] = testScores;
```

This statement defines three `int` variables named `score1`, `score2`, and `score3`. The variables are defined as `int`s because the elements of the `testScores` array are `int`s. After the statement executes, the `score1` variable will be assigned 80, the `score2` variable will be assigned 90, and the `score3` variable will be assigned 100.

The following code snippet shows how the array can be unpacked into variables that are used in a calculation:

```
int testScores[] = {80, 90, 100};
auto [score1, score2, score3] = testScores;
double average = (score1 + score2 + score3) / 3.0;
```

7.6 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

doubles).

Program 7-15

```
 1  // This program uses two parallel arrays: one for hours
 2  // worked and one for pay rate.
 3  #include
 4  #include
 5  using namespace std;
 6
 7  int main()
 8  {
 9      const int NUM_EMPLOYEES = 5;     // Number of employees
10      int hours[NUM_EMPLOYEES];        // Holds hours worked
11      double payRate[NUM_EMPLOYEES];   // Holds pay rates
12
13      // Input the hours worked and the hourly pay rate.
14      cout << "Enter the hours worked by " << NUM_EMPLOYEES
15           << " employees and their\n"
16           << "hourly pay rates.\n";
17      for (int index = 0; index < NUM_EMPLOYEES; index++)
18      {
19          cout << "Hours worked by employee #" << (index+1) << ": ";
20          cin >> hours[index];
21          cout << "Hourly pay rate for employee #" << (index+1) << ": ";
22          cin >> payRate[index];
23      }
24
25      // Display each employee's gross pay.
26      cout << "Here is the gross pay for each employee:\n";
27      cout << fixed << showpoint << setprecision(2);
28      for (int index = 0; index < NUM_EMPLOYEES; index++)
29      {
30          double grossPay = hours[index] * payRate[index];
31          cout << "Employee #" << (index + 1);
32          cout << ": $" << grossPay << endl;
33      }
34      return 0;
35  }
```

Program Output with Example Input Shown in Bold

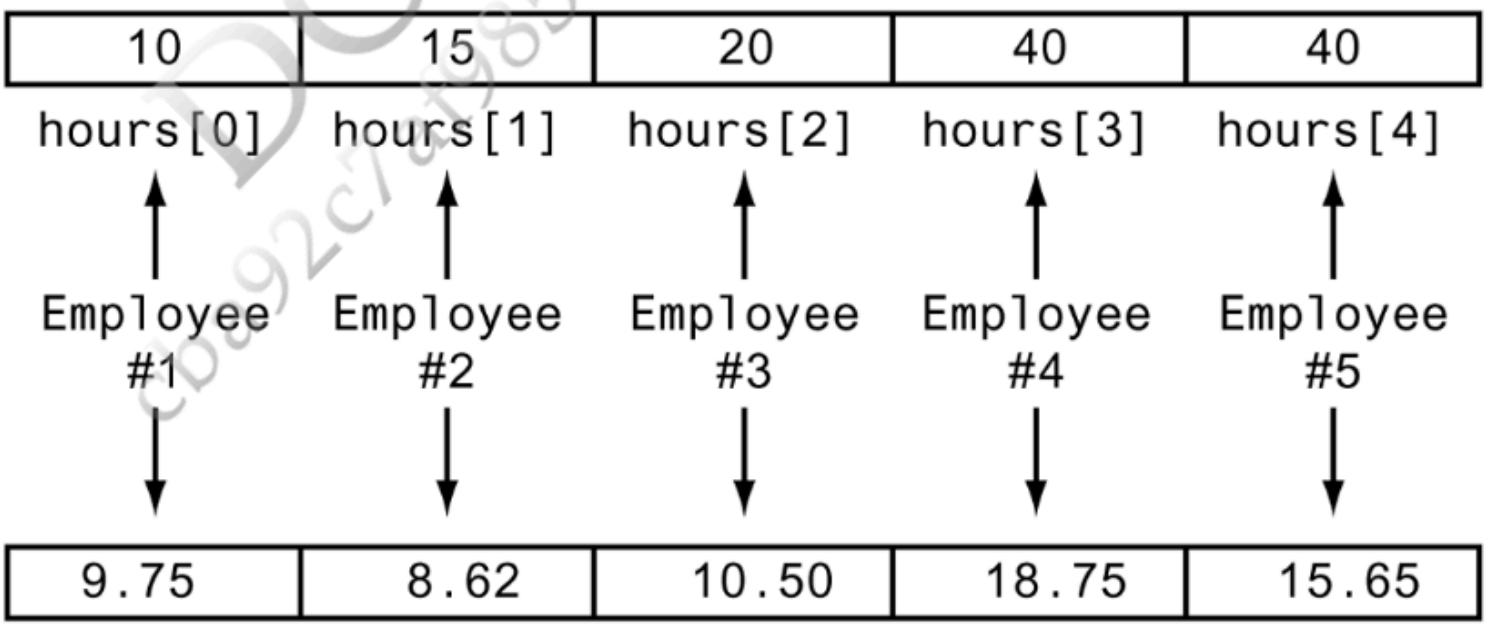
```
Enter the hours worked by 5 employees and their
hourly pay rates.
Hours worked by employee #1: 10 Enter
```

```
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 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 Parallel arrays



payRate[0] payRate[1] payRate[2] payRate[3] payRate[4]

Figure 7-13 Full Alternative Text

Checkpoint

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;
}
```

```
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;
}
```

7.7 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

```
1  // This program demonstrates that an array element is passed
2  // to a function like any other variable.
3  #include
4  using namespace std;
```

```
 4  using namespace std;
 5
 6  void showValue(int); // Function prototype
 7
 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 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
 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
```

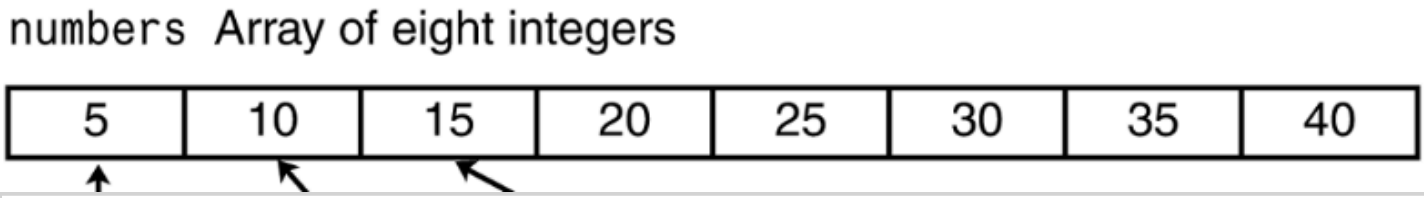
In Program 7-17, the function `showValues` is called in the following statement that appears in line 12:

```
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]` are displayed, it is actually the contents of `numbers[0]` that appear on the screen.

Figure 7-14 Relationship between the `numbers` array and the `nums` parameter



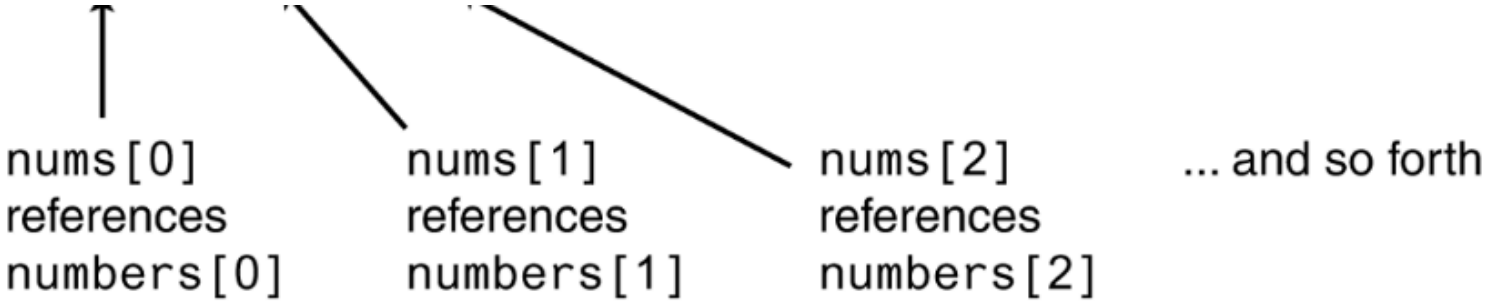


Figure 7-14 Full Alternative Text

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 A**.)

```
#include
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
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.
//
```

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

```
#include
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
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.
//
```

7.8 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 he or she will 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 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.

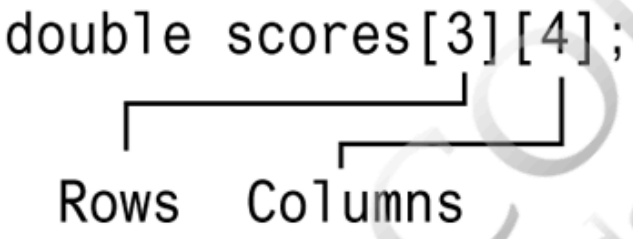
Figure 7-15 Two-dimensional array

	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]

Figure 7-15 Full Alternative Text

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:



7.8-3 Full Alternative Text

The first size declarator specifies the number of rows, and the second size declarator specifies the number of columns. Notice 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 step through each element of a two-dimensional array usually do so with nested loops. Program 7-21 is an example.

Program 7-21

```
1  // This program demonstrates a two-dimensional array.
2  #include
3  #include
4  using namespace std;
5
6  int main()
7  {
8      const int NUM_DIVS = 3;           // Number of divisions
9      const int NUM_QTRS = 4;           // Number of quarters
10     double sales[NUM_DIVS][NUM_QTRS]; // Array with 3 rows and 4 columns.
11     double totalSales = 0;            // To hold the total sales.
12     int div, qtr;                     // Loop counters
```

```
12      int div, qtr;                       // Loop counters.
13
14      cout << "This program will calculate the total sales of\n";
15      cout << "all the company's divisions.\n";
16      cout << "Enter the following sales information:\n\n";
17
18      // Nested loops to fill the array with quarterly
19      // sales figures for each division.
20      for (div = 0; div < NUM_DIVS; div++)
21      {
22          for (qtr = 0; qtr < NUM_QTRS; qtr++)
23          {
24              cout << "Division " << (div + 1);
25              cout << ", Quarter " << (qtr + 1) << ": $";
26              cin >> sales[div][qtr];
27          }
28          cout << endl; // Print blank line.
29      }
30
31      // Nested loops used to add all the elements.
32      for (div = 0; div < NUM_DIVS; div++)
33      {
34          for (qtr = 0; qtr < NUM_QTRS; qtr++)
35              totalSales += sales[div][qtr];
36      }
37
38      cout << fixed << showpoint << setprecision(2);
39      cout << "The total sales for the company are: $";
40      cout << totalSales << endl;
41      return 0;
42  }
```

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 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 Two-dimensional array initialization

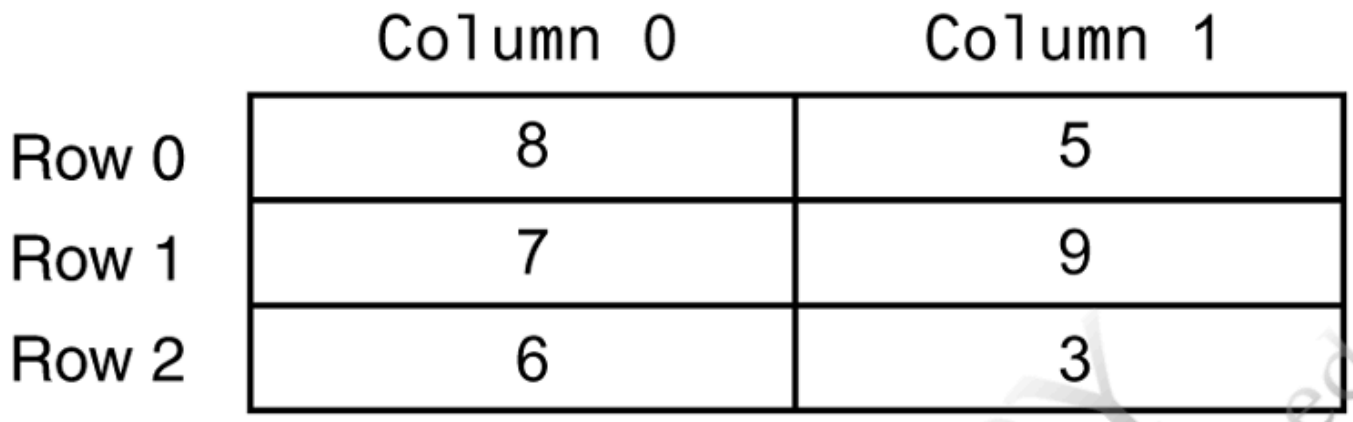


Figure 7-16 Full Alternative Text

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 that 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

```
 1  // This program demonstrates accepting a 2D array argument.
 2  #include
 3  #include
 4  using namespace std;
 5
 6  // Global constants
 7  const int COLS = 4;      // Number of columns in each array
 8  const int TBL1_ROWS = 3; // Number of rows in table1
 9  const int TBL2_ROWS = 4; // Number of rows in table2
10
11  void showArray(const int [][COLS], int); // Function prototype
12
13  int main()
14  {
15      int table1[TBL1_ROWS][COLS] = {{1, 2, 3, 4},
16                                     {5, 6, 7, 8},
17                                     {9, 10, 11, 12}};
18      int table2[TBL2_ROWS][COLS] = {{10, 20, 30, 40},
19                                     {50, 60, 70, 80},
20                                     {90, 100, 110, 120},
21                                     {130, 140, 150, 160}};
22
23      cout << "The contents of table1 are:\n";
24      showArray(table1, TBL1_ROWS);
25      cout << "The contents of table2 are:\n";
26      showArray(table2, TBL2_ROWS);
27      return 0;
28  }
29
30  //*************************************************************
31  // Function Definition for showArray                          *
32  //The first argument is a two-dimensional int array with COLS  *
33  // columns. The second argument, rows, specifies the number of *
34  // rows in the array. The function displays the array's contents.*
35  //*************************************************************
36
37  void showArray(const int numbers[][COLS], int rows)
38  {
39      for (int x = 0; x < rows; x++)
40      {
41          for (int y = 0; y < COLS; y++)
```

```
41          for (int y = 0; y < COLS; y++)
42          {
43              cout << setw(4) << numbers[x][y] << " ";
44          }
45          cout << endl;
46      }
47  }
```

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 Memory organization of a two-dimensional array

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
```

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

Notice 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 this:

```
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;
}
```

Notice 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 this:

```
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;
}
```

7.9 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.

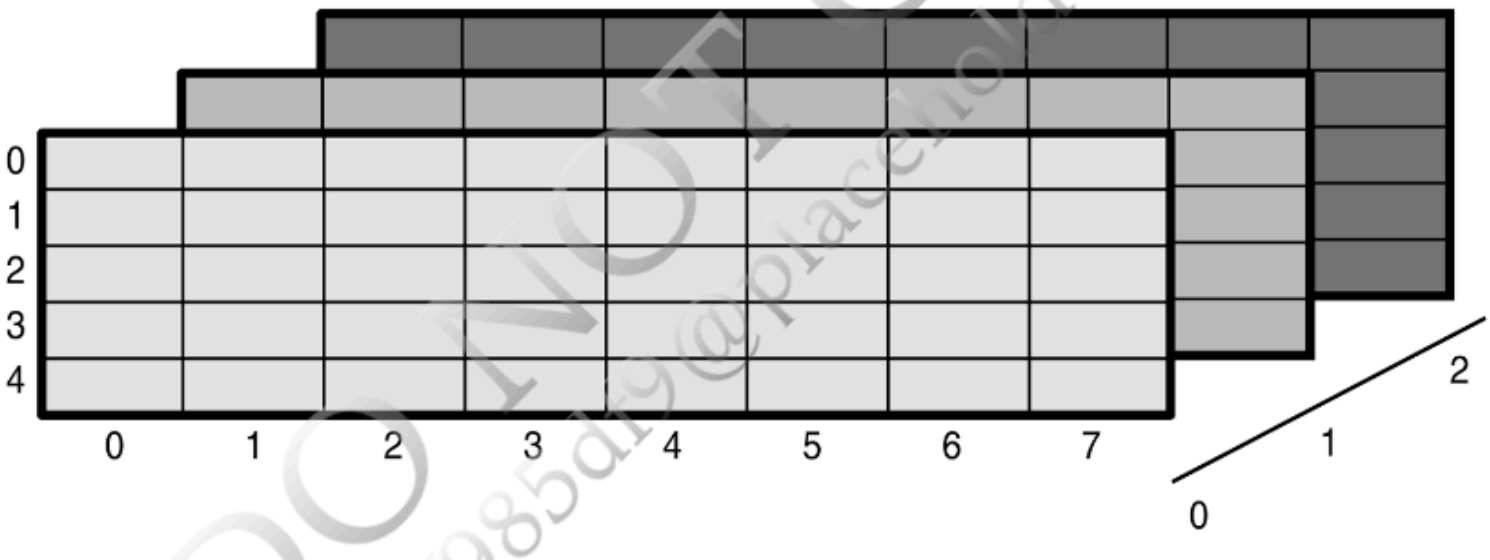
C++ allows you to create arrays with virtually any number of dimensions. Here is an example of a three-dimensional array definition:

```
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 A three-dimensional array



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 that 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.

7.24 Fill in the table below so that 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.

7.10 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 7-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
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.
```

```
47      return true; // If we make it this far, the values are the same.
48  }
```

Program Output

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

7.11 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 17.

In this section, you will learn to use the `vector` data type. A `vector` is a container that can store data. It 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.
- `vectors` can report the number of elements they contain.

Note

To use the `vector` data type, you should also have the `using namespace std;` statement in your program.

Defining a `vector`

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

```
#include
```

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 numbers;
```

This statement defines `numbers` as a `vector` of `ints`. Notice 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 numbers(10);
```

This statement defines `numbers` as a `vector` of 10 `ints`. 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 in 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 numbers(10, 2);
```

In this statement, `numbers` is defined as a `vector` of 10 `ints`. 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 `set1` is a `vector` of `ints` that already has values stored in it.

```
vector 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 `vector` Definitions

Definition Format	Description
`vector amounts;`	Defines `amounts` as an empty `vector` of `floats`.
`vector names;`	Defines `names` as an empty `vector` of `string` objects.
`vector scores(15);`	Defines `scores` as a `vector` of 15 `ints`.
`vector letters(25, 'A');`	Defines `letters` as a `vector` of 25 characters. Each element is initialized with `'A'`.
`vector 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`

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

```
vector numbers { 10, 20, 30, 40 };
```

This statement defines a `vector` of `int`s named `numbers`. The `vector` will have four elements, initialized with the values 10, 20, 30, and 40. Notice 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

```
 1  // This program stores, in two vectors, the hours worked by 5
 2  // employees, and their hourly pay rates.
 3  #include
 4  #include
 5  #include
 6  using namespace std;
 7
 8  int main()
 9  {
10      const int NUM_EMPLOYEES = 5;            // Number of employees
11      vector hours(NUM_EMPLOYEES);       // A vector of integers
12      vector payRate(NUM_EMPLOYEES);  // A vector of doubles
13      int index;                              // Loop counter
14
15      // Input the data.
16      cout << "Enter the hours worked by " << NUM_EMPLOYEES;
17      cout << " employees and their hourly rates.\n";
18      for (index = 0; index < NUM_EMPLOYEES; index++)
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 Program 7-24 uses the following statements in lines 11 and 12 to define two `vectors`:

```
vector hours(NUM_EMPLOYEES);      // A vector of integers
vector payRate(NUM_EMPLOYEES); // A vector of doubles
```

Both of the `vector`s 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 `vector`s:

```
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`

With C++ 11 and later, 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

```
 1  // This program demonstrates the range-based for loop with a vector.
 2  include
 3  #include
 4  using namespace std;
 5
 6  int main()
 7  {
 8      // Define and initialize a vector.
 9      vector numbers { 10, 20, 30, 40, 50 };
10
11      // Display the vector elements.
12      for (int val : numbers)
13          cout << val << endl;
14
```

```
14
15      return 0;
16  }
```

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
 3  #include
 4  using namespace std;
 5
 6  int main()
 7  {
 8      // Define a vector.
 9      vector 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
```

```
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 five elements. Notice 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 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 `vectors`, `hours` and `payRate`, are defined without starting sizes. Because these `vectors` have no starting elements, the `push_back` member function is used to store values in the `vectors`.

Program 7-27

```
 1  // This program stores, in two arrays, the hours worked by 5
 2  // employees, and their hourly pay rates.
 3  #include
 4  #include
 5  #include
 6  using namespace std;
 7
 8  int main()
 9  {
10      vector hours;      // hours is an empty vector
11      vector 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
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
```

```
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 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 `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 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

```
1  // This program demonstrates the vector size
2  // member function.
3  #include
4  #include
5  using namespace std;
```

```
 5  using namespace std;
 6
 7  // Function prototype
 8  void showValues(vector);
 9
10  int main()
11  {
12      vector values;
13
14      // Put a series of numbers in the vector.
15      for (int count = 0; count < 7; count++)
16          values.push_back(count * 2);
17
18      // Display the numbers.
19      showValues(values);
20      return 0;
21  }
22
23  //*************************************************
24  // Definition of function showValues.             *
25  // This function accepts an int vector as its     *
26  // argument. The value of each of the vector's    *
27  // elements is displayed.                         *
28  //*************************************************
29
30    void showValues(vector vect)
31    {
32       for (int count = 0; count < vect.size(); count++)
33          cout << vect[count] << endl;
34    }
```

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 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

```
1  // This program demonstrates the vector pop_back member function.
2  #include
3  #include
4  using namespace std;
5
6  int main()
7  {
8      vector values;
9
10     // Store values in the vector.
11     values.push_back(1);
12     values.push_back(2);
13     values.push_back(3);
14     cout << "The size of values is " << values.size() << endl;
15
16     // Remove a value from the vector.
17     cout << "Popping a value from the vector ... \n";
18     values.pop_back();
19     cout << "The size of values is now " << values.size() << endl;
20
21     // Now remove another value from the vector.
22     cout << "Popping a value from the vector ... \n";
23     values.pop_back();
24     cout << "The size of values is now " << values.size() << endl;
25
26     // Remove the last value from the vector.
27     cout << "Popping a value from the vector ... \n";
28     values.pop_back();
29     cout << "The size of values is now " << values.size() << endl;
30     return 0;
31 }
```

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

```
1  // This program demonstrates the vector clear member function.
2  #include
3  #include
4  using namespace std;
5
6  int main()
7  {
8      vector values(100);
9
10     cout << "The values vector has "
11          << values.size() << " elements.\n";
12     cout << "I will call the clear member function ... \n";
13     values.clear();
14     cout << "Now, the values vector has "
15          << values.size() << " elements.\n";
16    return 0;
17 }
```

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

```
 1  // This program demonstrates the vector's empty member function.
 2  #include
 3  #include
 4  using namespace std;
 5
 6  // Function prototype
 7  double avgVector(vector);
 8
 9  int main()
10  {
11      vector values;  // A vector to hold values
12      int numValues;      // The number of values
13      double average;     // To hold the average
14
15      // Get the number of values to average.
16      cout << "How many values do you wish to average? ";
17      cin >> numValues;
18
19      // Get the values and store them in the vector.
20      for (int count = 0; count < numValues; count++)
21      {
22          int tempValue;
23          cout << "Enter a value: ";
24          cin >> tempValue;
25          values.push_back(tempValue);
26      }
27
28      // Get the average of the values and display it.
29      average = avgVector(values);
30      cout << "Average: " << average << endl;
```

```
31      return 0;
32  }
33
34  //*************************************************************
35  // Definition of function avgVector.                          *
36  // This function accepts an int vector as its argument. If    *
37  // the vector contains values, the function returns the       *
38  // average of those values. Otherwise, an error message is    *
39  // displayed and the function returns 0.0.                    *
40  //*************************************************************
41
42  double avgVector(vector vect)
43  {
44      int total = 0;     // accumulator
45      double avg;        // average
46
47      if (vect.empty())  // Determine if the vector is empty
48      {
49          cout << "No values to average.\n";
50          avg = 0.0;
51      }
52      else
53      {
54          for (int count = 0; count < vect.size(); count++)
55              total += vect[count];
56          avg = total / vect.size();
57      }
58      return avg;
59  }
```

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 Some of the `vector` Member Functions

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`.
`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 *`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`.
`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`.

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 4 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.

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 ten 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 ten 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.

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 Errors

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[220];
```

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 numbers = { 1, 2, 3, 4 };
```