Multidimensional Arrays

Some data fits in lists, such as the data discussed in the previous two chapters, and other data is better suited for tables of information. This chapter takes arrays one step further. The previous chapters introduced single-dimensional arrays; arrays that have only one subscript and represent lists of values.

This chapter introduces arrays of more than one dimension, called *multidimensional arrays*. Multidimensional arrays, sometimes called *tables* or *matrices*, have at least two dimensions (rows and columns). Many times they have more than two.

This chapter introduces the following concepts:

- Multidimensional arrays
- Reserving storage for multidimensional arrays
- Putting data in multidimensional arrays
- Using nested `for` loops to process multidimensional arrays

If you understand single-dimensional arrays, you should have no trouble understanding arrays that have more than one dimension.

Multidimensional Array Basics

A multidimensional array has more than one subscript.

A multidimensional array is an array with more than one subscript. Whereas a single-dimensional array is a list of values, a multidimensional array simulates a table of values, or multiple tables of values. The most commonly used table is a two-dimensional table (an array with two subscripts).

Suppose a softball team wanted to keep track of its players' batting records. The team played 10 games, and there are 15 players on the team. Table 25.1 shows the team's batting record.

Table 25.1. A softball team's batting record.

Player Name	*Game 1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
Adams	2	1	0	0	2	3	3	1	1	2
Berryhill	1	0	3	2	5	1	2	2	1	0
Downing	1	0	2	1	0	0	0	0	2	0
Edwards	0	3	6	4	6	4	5	3	6	3
Franks	2	2	3	2	1	0	2	3	1	0
Grady	1	3	2	0	1	5	2	1	2	1
Howard	3	1	1	1	2	0	1	0	4	3
Jones	2	2	1	2	4	1	0	7	1	0
Martin	5	4	5	1	1	0	2	4	1	5
Powers	2	2	3	1	0	2	1	3	1	2
Smith	1	1	2	1	3	4	1	0	3	2
Smithtown	1	0	1	2	1	0	3	4	1	2
Townsend	0	0	0	0	0	0	1	0	0	0
Ulmer	2	2	2	2	2	1	1	3	1	3
Williams	2	3	1	0	1	2	1	2	0	3

Do you see that the softball table is a two-dimensional table? It has rows (the first dimension) and columns (the second dimension). Therefore, this is called a two-dimensional table with 15 rows and 10 columns. (Generally, the number of rows is specified first.)

Each row has a player's name, and each column has a game number associated with it, but these are not part of the actual data. The data consists of only 150 values (15 rows by 10 columns). The data in a two-dimensional table always is the same type of data; in this case, every value is an integer. If it were a table of salaries, every element would be a floating-point decimal.

The number of dimensions, in this case two, corresponds to the dimensions in the physical world. The single-dimensioned array is a line, or list of values. Two dimensions represent both length and width. You write on a piece of paper in two dimensions; two dimensions represent a flat surface. Three dimensions represent width, length, and depth. You have seen 3-D movies. Not only do the images have width and height, but they also seem to have depth. Figure 25.1 shows what a three-dimensional array looks like if it has a depth of four, six rows, and three columns. Notice that a three-dimensional table resembles a cube.

A three-dimensional table has three dimensions: depth, rows, and columns.

It is difficult to visualize more than three dimensions. However, you can think of each dimension after three as another occurrence. In other words, a list of one player's season batting record can be stored in an array. The team's batting record (as shown in Table 25.1) is two-dimensional. The league, made of up several teams' batting records, represents a three-dimensional table. Each team (the depth of the table) has rows and columns of batting data. If there is more than one league, it is another dimension (another set of data).

C++ enables you to store several dimensions, although "real-world" data rarely requires more than two or three.

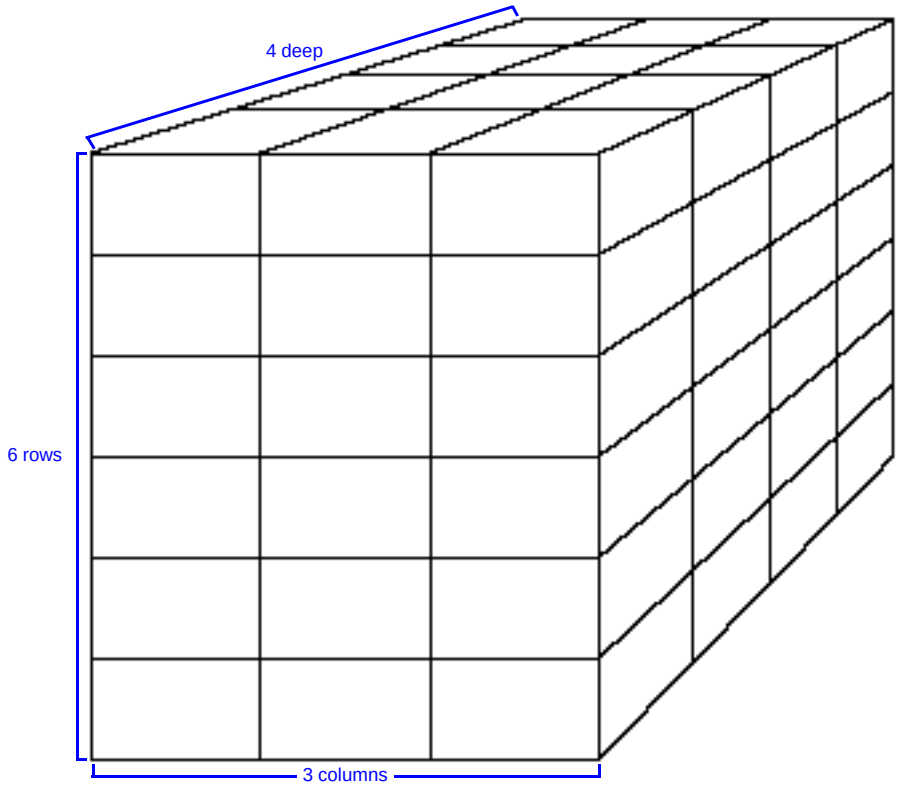


Figure 25.1. Representing a three-dimensional table (a cube).

Reserving Multidimensional Arrays

When you reserve a multidimensional array, you must inform C++ that the array has more than one dimension by putting more than one subscript in brackets after the array name. You must put a separate number, in brackets, for each dimension in the table. For example, to reserve the team data from Table 25.1, you use the following multidimensional array declaration.

Declare an integer array called `teams` *with 15 rows and 10 columns.*

```
int teams[15][10];  // Reserves a two-dimensional table.
```

CAUTION: Unlike other programming languages, C++ requires you to enclose each dimension in brackets. Do not reserve multidimensional array storage like this:

```
int teams[15,10];   // Invalid table declaration.
```

Properly reserving the `teams` table produces a table with 150 elements. Figure 25.2 shows what each element's subscript looks like.

columns

rows

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]	[0][5]	[0][6]	[0][7]	[0][8]	[0][9]
[1][0]	[1][1]	[1][2]	[1][3]	[1][4]	[1][5]	[1][6]	[1][7]	[1][8]	[1][9]
[2][0]	[2][1]	[2][2]	[2][3]	[2][4]	[2][5]	[2][6]	[2][7]	[2][8]	[2][9]
[3][0]	[3][1]	[3][2]	[3][3]	[3][4]	[3][5]	[3][6]	[3][7]	[3][8]	[3][9]
[4][0]	[4][1]	[4][2]	[4][3]	[4][4]	[4][5]	[4][6]	[4][7]	[4][8]	[4][9]
[5][0]	[5][1]	[5][2]	[5][3]	[5][4]	[5][5]	[5][6]	[5][7]	[5][8]	[5][9]
[6][0]	[6][1]	[6][2]	[6][3]	[6][4]	[6][5]	[6][6]	[6][7]	[6][8]	[6][9]
[7][0]	[7][1]	[7][2]	[7][3]	[7][4]	[7][5]	[7][6]	[7][7]	[7][8]	[7][9]
[8][0]	[8][1]	[8][2]	[8][3]	[8][4]	[8][5]	[8][6]	[8][7]	[8][8]	[8][9]
[9][0]	[9][1]	[9][2]	[9][3]	[9][4]	[9][5]	[9][6]	[9][7]	[9][8]	[9][9]
[10][0]	[10][1]	[10][2]	[10][3]	[10][4]	[10][5]	[10][6]	[10][7]	[10][8]	[10][9]
[11][0]	[11][1]	[11][2]	[11][3]	[11][4]	[11][5]	[11][6]	[11][7]	[11][8]	[11][9]
[12][0]	[12][1]	[12][2]	[12][3]	[12][4]	[12][5]	[12][6]	[12][7]	[12][8]	[12][9]
[13][0]	[13][1]	[13][2]	[13][3]	[13][4]	[13][5]	[13][6]	[13][7]	[13][8]	[13][9]
[14][0]	[14][1]	[14][2]	[14][3]	[14][4]	[14][5]	[14][6]	[14][7]	[14][8]	[14][9]

Figure 25.2. Subscripts for the softball team table.

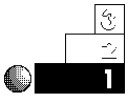
If you had to track three teams, each with 15 players and 10 games, the three-dimensional table would be created as follows:

```
int teams[3][15][10]; // Reserves a three-dimensional table.
```

The far-right dimension always represents columns, the next represents rows, and so on.

When creating a two-dimensional table, always put the maximum number of rows first, and the maximum number of columns second. C++ always uses `0` as the starting subscript of each dimension. The last element, the lower-right element of the `teams` table, is `teams[2][14][9]`.

Examples



1. Suppose you wanted to keep track of utility bills for the year. You can store 12 months of four utilities in a two-dimensional table of floating-point amounts, as the following array declaration demonstrates:

   ```
   float utilities[12][4];          // Reserves 48 elements.
   ```

   You can compute the total number of elements in a multidimensional array by multiplying the subscripts. Because 12 times 4 is 48, there are 48 elements in this array (12 rows, 4 columns). Each of these elements is a floating-point data type.

2. If you were keeping track of five years' worth of utilities, you have to add an extra dimension. The first dimension is the years, the second is the months, and the last is the individual utilities. Here is how you reserve storage:

   ```
   float utilities[5][12][4];      // Reserves 240 elements.
   ```

Mapping Arrays to Memory

C++ approaches multidimensional arrays a little differently than most programming languages do. When you use subscripts, you do not have to understand the internal representation of multidimensional arrays. However, most C++ programmers think a deeper understanding of these arrays is important, especially when programming advanced applications.

A two-dimensional array is actually an *array of arrays*. You program multidimensional arrays as though they were tables with rows and columns. A two-dimensional array is actually a single-dimensional array, but each of its elements is not an integer, floating-point, or character, but another array.

Knowing that a multidimensional array is an array of other arrays is critical when passing and receiving such arrays. C++ passes all arrays, including multidimensional arrays, by address. Suppose you were using an integer array called `scores`, reserved as a 5-by-6 table. You can pass `scores` to a function called `print_it()`, as follows:

```
print_it(scores);          // Passes table to a function.
```

The function `print_it()` has to identify the type of parameter being passed to it. The `print_it()` function also must recognize that the parameter is an array. If `scores` were one-dimensional, you could receive it as

```
print_it(int scores[])          // Works only if scores
                                // is one-dimensional.
```

or

```
print_it(int scores[10])        // Assuming scores
                                // has 10 elements.
```

If `scores` were a multidimensional table, you would have to designate each pair of brackets and put the maximum number of subscripts in its brackets, as in

```
print_it(int scores[5][6])       // Inform print_it() of
                                 // the array's dimensions.
```

or

```
print_it(int scores[][6])        // Inform print_it() of
                                 // the array's dimensions.
```

Notice you do not have to explicitly state the maximum subscript on the first dimension when receiving multidimensional

arrays, but you must designate the second. If `scores` were a three-dimensional table, dimensioned as 10 by 5 by 6, you would receive it with `print_it()` as

```
print_it(int scores[][5][6])      // Only first dimension
                                  // is optional.
```

or

```
print_it(int scores[10][5][6])     // Inform print_it() of
                                   // array's dimensions.
```

You should not have to worry too much about the way tables are physically stored. Even though a two-dimensional table is actually an array of arrays (and each of those arrays contains another array if it is a three-dimensional table), you can use subscripts to program multidimensional arrays as if they were stored in row-and-column order.

C++ stores multidimensional arrays in row order.

Multidimensional arrays are stored in *row order*. Suppose you want to keep track of a 3-by-4 table. The top of Figure 25.3 shows how that table (and its subscripts) are visualized. Despite the two-dimensional table organization, your memory is still sequential storage. C++ has to map multidimensional arrays to single-dimensional memory, and it does so in row order.

Each row fills memory before the next row is stored. Figure 25.3 shows how a 3-by-4 table is mapped to memory.

The entire first row (`table[0][0]` through `table[0][3]`) is stored first in memory before any of the second row. A table is actually an array of arrays, and, as you learned in previous chapters, array elements are always stored sequentially in memory. Therefore, the first row (array) completely fills memory before the second row. Figure 25.3 shows how two-dimensional arrays map to memory.

Defining Multidimensional Arrays

C++ is not picky about the way you define a multidimensional array when you initialize it at declaration time. As with single-dimensional arrays, you initialize multidimensional arrays with

braces that designate dimensions. Because a multidimensional array is an array of arrays, you can nest braces when you initialize them.

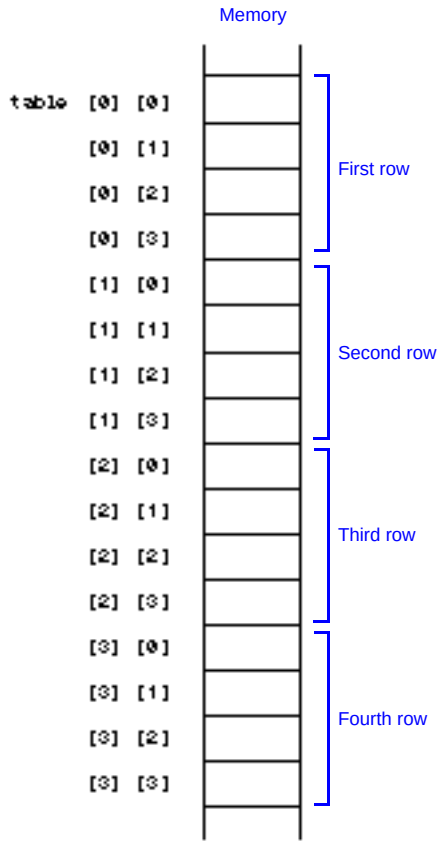


Figure 25.3. Mapping a two-dimensional table to memory.

The following three array definitions fill the three arrays `ara1`, `ara2`, and `ara3`, as shown in Figure 25.4:

```
int ara1[5] = {8, 5, 3, 25, 41};  // One-dimensional array.
int ara2[2][4]={{4, 3, 2, 1},{1, 2, 3, 4}};
int ara3[3][4]={{1, 2, 3, 4},{5, 6, 7, 8},{9, 10, 11, 12}};
```

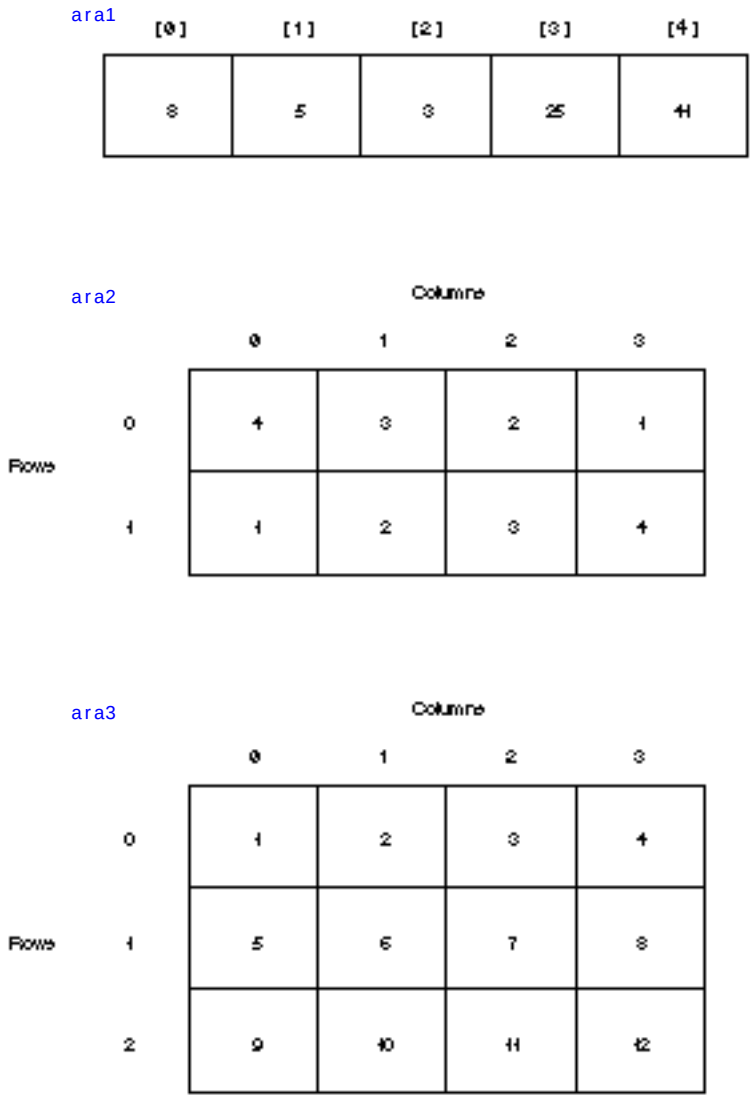


Figure 25.4. After initializing a table.

Notice that the multidimensional arrays are stored in row order. In `ara3`, the first row receives the first four elements of the definition (1, 2, 3, and 4).

TIP: To make a multidimensional array initialization match the array's subscripts, some programmers like to show how arrays are filled. Because C++ programs are free-form, you can initialize ara2 and ara3 as

```
int ara2[2][4]={{4, 3, 2, 1},  // Does exactly the same
                {1, 2, 3, 4}}; // thing as before.

int ara3[3][4]={{1, 2, 3, 4},
                {5, 6, 7, 8},
                {9, 10, 11, 12}; // Visually more
                                 // obvious.
```

You can initialize a multidimensional array as if it were single-dimensional in C++. You must keep track of the row order if you do this. For instance, the following two definitions also reserve storage for and initialize ara2 and ara3:

```
int ara2[2][4]={4, 3, 2, 1, 1, 2, 3, 4};
int ara3[3][4]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
```

There is no difference between initializing ara2 and ara3 with or without the nested braces. The nested braces seem to show the dimensions and how C++ fills them a little better, but the choice of using nested braces is yours.

TIP: Multidimensional arrays (unless they are global) are not initialized to specific values unless you assign them values at declaration time or in the program. As with single-dimensional arrays, if you initialize one or more of the elements, but not all of them, C++ fills the rest with zeros. If you want to fill an entire multidimensional array with zeros, you can do so with the following:

```
float sales[3][4][7][2] = {0};  // Fills all sales
                                // with zeros.
```

One last point to consider is how multidimensional arrays are viewed by your compiler. Many people program in C++ for years, but never understand how tables are stored internally. As long as you use subscripts, a table's internal representation should not matter. When you learn about pointer variables, however, you might want to know how C++ stores your tables in case you want to reference them with pointers (as shown in the next few chapters).

Figure 25.5 shows the way C++ stores a 3-by-4 table in memory. Unlike single-dimensional arrays, each element is stored contiguously, but notice how C++ views the data. Because a table is an array of arrays, the array name contains the address of the start of the primary array. Each of those elements points to the arrays it contains (the data in each row). This coverage of table storage is for your information only, at this point. As you become more proficient in C++, and write more powerful programs that manipulate internal memory, you might want to review this table storage method.

Tables and **for** Loops

As the following examples show, nested `for` loops are useful when you want to loop through every element of a multidimensional table.

For instance, the section of code,

```
for (row=0; row<2; row++)
   { for (col=0; col<3; col++)
        { cout << row << " " << col "\n"; }
   }
```

produces the following output:

```
0    0
0    1
0    2
1    0
1    1
1    2
```

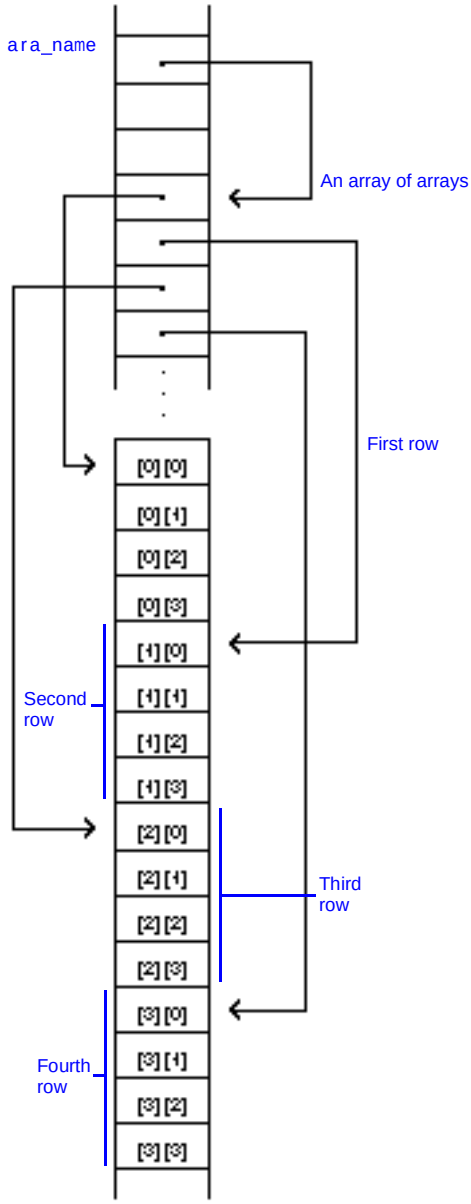


Figure 25.5. Internal representation of a two-dimensional table.

These numbers are the subscripts, in row order, for a two-row by three-column table dimensioned with

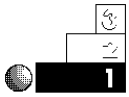
```
int table[2][3];
```

Nested loops work well with multi-dimensional arrays.

Notice there are as many `for` loops as there are subscripts in the array (two). The outside loop represents the first subscript (the rows), and the inside loop represents the second subscript (the columns). The nested `for` loop steps through each element of the table.

You can use `cin`, `gets()`, `get`, and other input functions to fill a table, and you also can assign values to the elements when declaring the table. More often, the data comes from data files on the disk. Regardless of what method stores the values in multidimensional arrays, nested `for` loops are excellent control statements to step through the subscripts. The following examples demonstrate how nested `for` loops work with multidimensional arrays.

Examples



1. The following statements reserve enough memory elements for a television station's ratings (*A* through *D*) for one week:

   ```
   char ratings[7][48];
   ```

   These statements reserve enough elements to hold seven days (the rows) of ratings for each 30-minute time slot (48 of them in a day).

   Every element in a table is always the same type. In this case, each element is a character variable. Some are initialized with the following assignment statements:

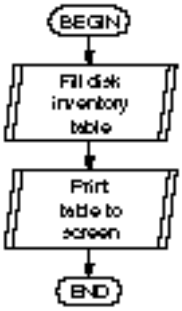
   ```
   shows[3][12] = 'B';      // Stores B in 4th row, 13th column.
   shows[1][5] = 'A' ;      // Stores C in 2nd row, 6th column.
   shows[6][20] = getch();  // Stores the letter the user types.
   ```

2. A computer company sells two sizes of disks: 3 1/2-inch and 5 1/4-inch. Each disk comes in one of four capacities: single-sided double-density, double-sided double-density, single-sided high-density, and double-sided high-density.

The disk inventory is well-suited for a two-dimensional table. The company determined that the disks have the following retail prices:

	Double Density		***High Density***	
	Single	*Double*	*Single*	*Double*
3 1/2-*inch*	2.30	2.75	3.20	3.50
5 1/4-*inch*	1.75	2.10	2.60	2.95

The company wants to store the price of each disk in a table for easy access. The following program stores the prices with assignment statements.




```
// Filename: C25DISK1.CPP
// Assigns disk prices to a table.
#include <iostream.h>
#include <iomanip.h>
void main()
{
   float disks[2][4];  // Table of disk prices.
   int row, col;       // Subscript variables.

   disks[0][0] = 2.39;      // Row 1, column 1
   disks[0][1] = 2.75;      // Row 1, column 2
   disks[0][2] = 3.29;      // Row 1, column 3
   disks[0][3] = 3.59;      // Row 1, column 4
   disks[1][0] = 1.75;      // Row 2, column 1
   disks[1][1] = 2.19;      // Row 2, column 2
   disks[1][2] = 2.69;      // Row 2, column 3
   disks[1][3] = 2.95;      // Row 2, column 4

   // Print the prices.
   for (row=0; row<2; row++)
      { for (col=0; col<4; col++)
        { cout << "$" << setprecision(2) <<
                  disks[row][col] << "\n"; }
      }

   return;
}
```

This program displays the prices as follows:

```
$2.39
$2.75
$3.29
$3.59
$1.75
$2.19
$2.69
$2.95
```

It prints them one line at a time, without any descriptive titles. Although the output is not labeled, it illustrates how you can use assignment statements to initialize a table, and how nested `for` loops can print the elements.

3. The preceding disk inventory would be displayed better if the output had descriptive titles. Before you add titles, it is helpful for you to see how to print a table in its native row and column format.

   Typically, you use a nested `for` loop, such as the one in the previous example, to print rows and columns. You should not output a newline character with every `cout`, however. If you do, you see one value per line, as in the previous program's output, which is not the row and column format of the table.

   You do not want to see every disk price on one line, but you want each row of the table printed on a separate line. You must insert a `cout << "\n";` to send the cursor to the next line each time the row number changes. Printing newlines after each row prints the table in its row and column format, as this program shows:

```
// Filename: C25DISK2.CPP
// Assigns disk prices to a table
// and prints them in a table format.
#include <iostream.h>
#include <iomanip.h>
void main()
{
```

```
   float disks[2][4];  // Table of disk prices.
   int row, col;

   disks[0][0] = 2.39;      // Row 1, column 1
   disks[0][1] = 2.75;      // Row 1, column 2
   disks[0][2] = 3.29;      // Row 1, column 3
   disks[0][3] = 3.59;      // Row 1, column 4
   disks[1][0] = 1.75;      // Row 2, column 1
   disks[1][1] = 2.19;      // Row 2, column 2
   disks[1][2] = 2.69;      // Row 2, column 3
   disks[1][3] = 2.95;      // Row 2, column 4

   // Print the prices
   for (row=0; row<2; row++)
      { for (col=0; col<4; col++)
        { cout << "$" << setprecision(2) <<
                  disks[row][col] << "\t";
        }

        cout << "\n";  // Prints a new line after each row.
      }

   return;
}
```

Here is the output of the disk prices in their native table order:

```
$2.39    $2.75    $3.29    $3.59
$1.75    $2.19    $2.69    $2.95
```

4. To add the titles, simply print a row of titles before the first row of values, then print a new column title before each column, as shown in the following program:

```
// Filename: C25DISK3.CPP
// Assigns disk prices to a table
// and prints them in a table format with titles.
#include <iostream.h>
#include <iomanip.h>
```

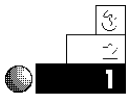
```
void main()
{
   float disks[2][4];  // Table of disk prices.
   int row, col;

   disks[0][0] = 2.39;      // Row 1, column 1
   disks[0][1] = 2.75;      // Row 1, column 2
   disks[0][2] = 3.29;      // Row 1, column 3
   disks[0][3] = 3.59;      // Row 1, column 4
   disks[1][0] = 1.75;      // Row 2, column 1
   disks[1][1] = 2.19;      // Row 2, column 2
   disks[1][2] = 2.69;      // Row 2, column 3
   disks[1][3] = 2.95;      // Row 2, column 4

   // Print the column titles.
   cout << "\tSingle-sided\tDouble-sided\tSingle-sided\t" <<
           "Double-sided\n";
   cout << "\tDouble-density\tDouble-density\tHigh-density" <<
           "\tHigh-density\n";

   // Print the prices
   for (row=0; row<2; row++)
      { if (row == 0)
       { cout << "3-1/2\"\t"; }         // Need \" to
                                       // print quotation.
     else
       { cout << "5-1/4\"\t"; }
     for (col=0; col<4; col++)    // Print the current row.
     { cout << setprecision(2) << "$" << disks[row][col]
            << "\t\t";
        }
        cout << "\n";    // Print a newline after each row.
      }

   return;
}
```

Here is the output from this program:

```
          Single-sided    Double-sided    Single-sided    Double-sided
          Double-density  Double-density  High-density    High-density
3-1/2"    $2.39           $2.75           $3.29           $3.59
5-1/4"    $1.75           $2.19           $2.69           $2.95
```

Review Questions

The answers to the review questions are in Appendix B.



1. What statement reserves a two-dimensional table of integers called `scores` with five rows and six columns?
2. What statement reserves a three-dimensional table of four character arrays called `initials` with 10 rows and 20 columns?
3. In the following statement, which subscript (first or second) represents rows and which represents columns?

```
int weights[5][10];
```

4. How many elements are reserved with the following statement?

```
int ara[5][6];
```

5. The following table of integers is called `ara`:

4	1	3	5	9
10	2	12	1	6
25	42	2	91	8

What values do the following elements contain?

a. `ara[2][2]`

b. `ara[0][1]`

c. `ara[2][3]`

d. `ara[2][4]`

6. What control statement is best for stepping through multidimensional arrays?
7. Notice the following section of a program:

```
int grades[3][5] = {80,90,96,73,65,67,90,68,92,84,70,
                    55,95,78,100};
```

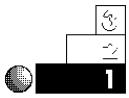
What are the values of the following:

a. `grades[2][3]`

b. `grades[2][4]`

c. `grades[0][1]`

Review Exercises



1. Write a program that stores and prints the numbers from 1 to 21 in a 3-by-7 table. (*Hint:* Remember C++ begins subscripts at 0.)
2. Write a program that reserves storage for three years' worth of sales data for five salespeople. Use assignment statements to fill the table with data, then print it, one value per line.

3. Instead of using assignment statements, use the `cin` function to fill the salespeople data from Exercise 2.
4. Write a program that tracks the grades for five classes, each having 10 students. Input the data using the `cin` function. Print the table in its native row and column format.

Summary

You now know how to create, initialize, and process multidimensional arrays. Although not all data fits in the compact format of tables, much does. Using nested `for` loops makes stepping through a multidimensional array straightforward.

EXAMPLE

One of the limitations of a multidimensional array is that each element must be the same data type. This keeps you from being able to store several kinds of data in tables. Chapter 28, “Structures,” shows you how to store data in different ways to overcome this limitation.