Chapter 9: Arrays and Strings

In this chapter you will learn about:

- Two- and Multidimensional Arrays
  - Accessing Array Components
  - Initialization During Declaration
  - Processing two-dimensional arrays
    - Initialization
    - Print
    - Input
    - Sum by Row
    - Sum by Column
    - Largest Element in Each Row and Each Column

Two-dimensional array: A collection of a fixed number of components arranged in rows and columns (that is, in two dimensions), wherein all components are of the same type.

The syntax for declaring a two-dimensional array is:

```
dataType  arrayName[intExp1][intExp2];
```

wherein `intExp1` and `intExp2` are constant expressions yielding positive integer values. The two expressions, `intExp1` and `intExp2`, specify the number of rows and the number of columns, respectively, in the array.

The statement:

```
double sales[10][5];
```

declares a two-dimensional array `sales` of 10 rows and 5 columns, in which every component is of type `double`. As in the case of a one-dimensional array, the rows are numbered `0...9` and the columns are numbered `0...4` (see Figure 9-10).

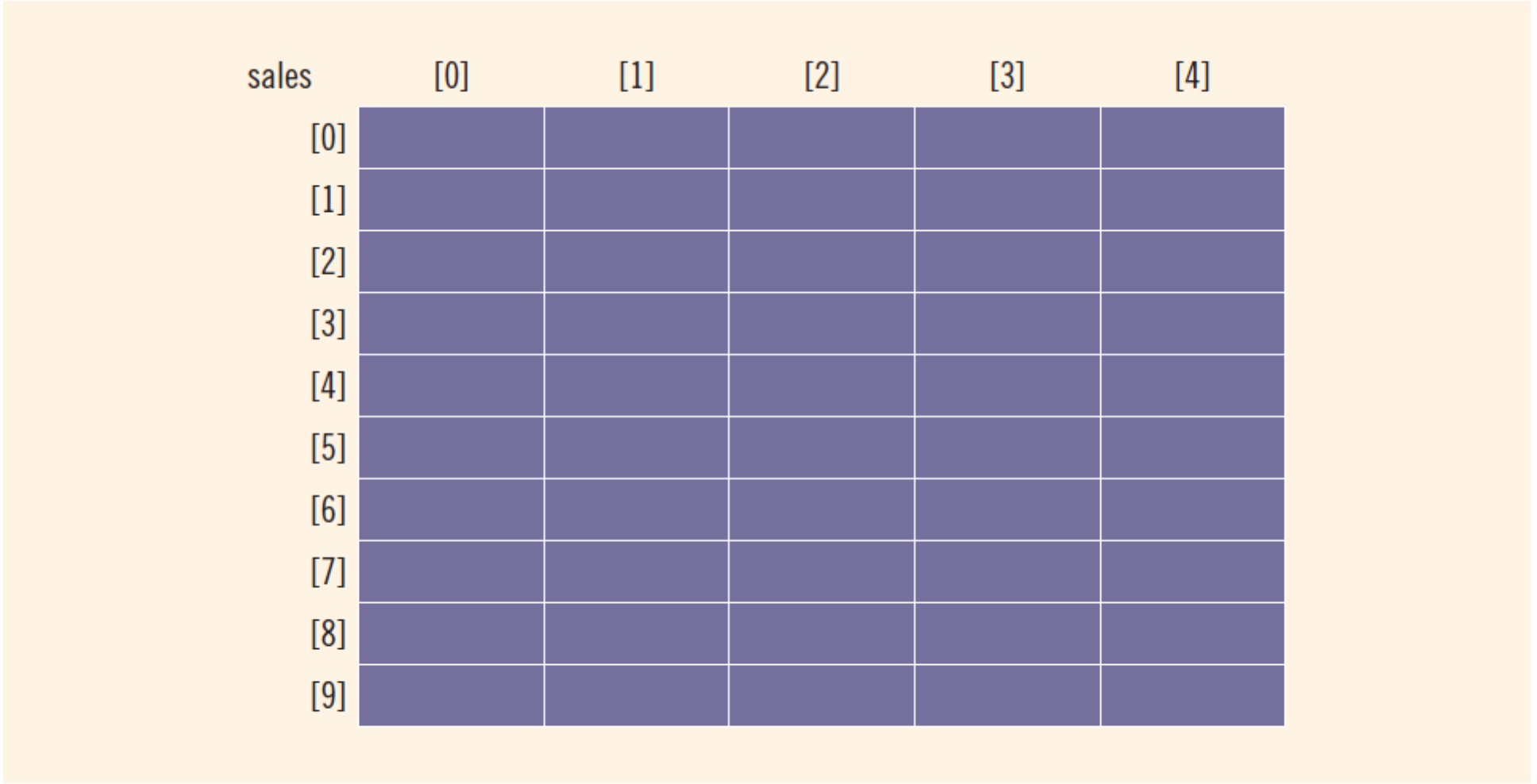


FIGURE 9-10 Two-dimensional array `sales`

Accessing Array Components

To access the components of a two-dimensional array, you need a pair of indices: one for the row position and one for the column position.

The syntax to access a component of a two-dimensional array is:

```
arrayName[indexExp1][indexExp2]
```

wherein `indexExp1` and `indexExp2` are expressions yielding nonnegative integer values. `indexExp1` specifies the row position; `indexExp2` specifies the column position.

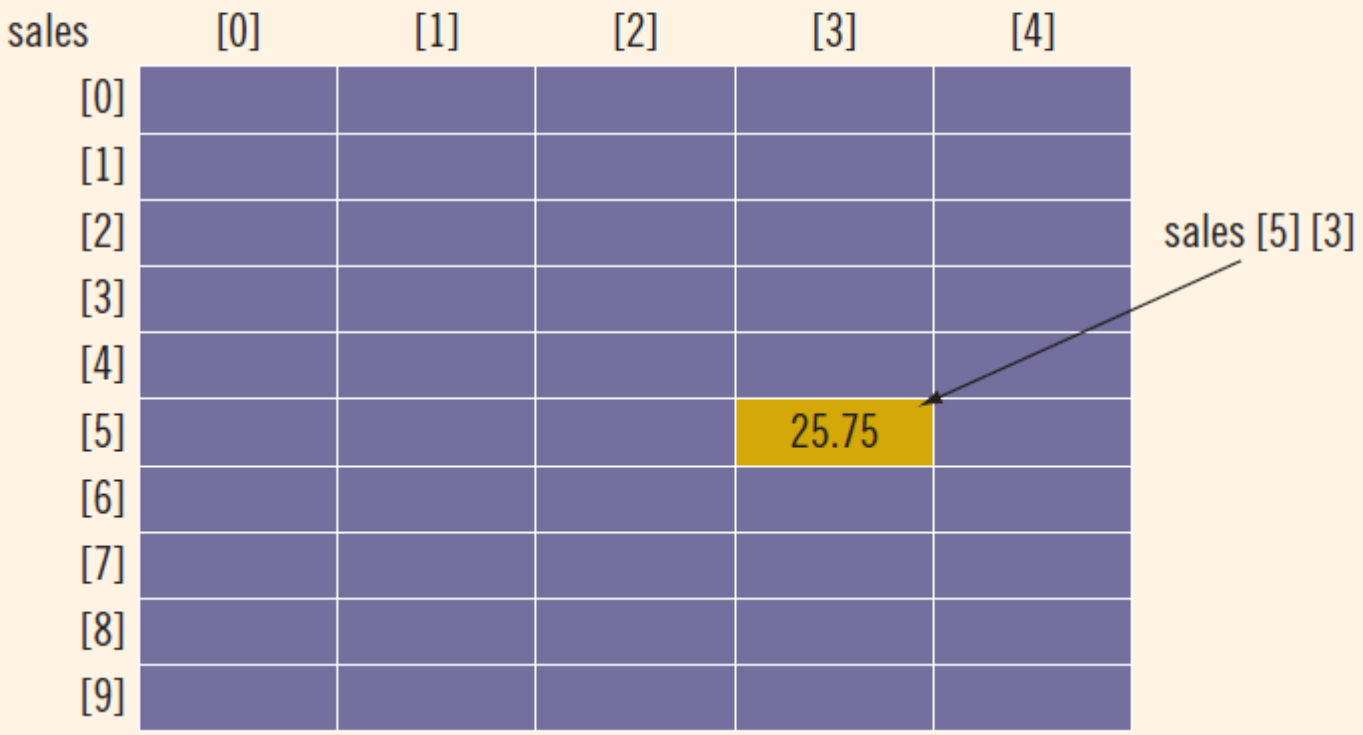
The statement:

```
sales[5][3] = 25.75;
```

stores `25.75` into row number `5` and column number `3` (that is, the sixth row and the fourth column) of the array `sales` (see Figure 9-11).

```
sales[5][3] = 25.75;
```

stores `25.75` into row number `5` and column number `3` (that is, the sixth row and the fourth column) of the array `sales` (see Figure 9-11).



Suppose that:

```
int i = 5;
int j = 3;
```

Then, the previous statement:

```
sales[5][3] = 25.75;
```

is equivalent to:

```
sales[i][j] = 25.75;
```

So the indices can also be variables.

Two-Dimensional Array Initialization During Declaration

Like one-dimensional arrays, two-dimensional arrays can be initialized when they are declared. The following example helps illustrate this concept. Consider the following statement:

```
int board[4][3] = {{2, 3, 1},
                   {15, 25, 13},
                   {20, 4, 7},
                   {11, 18, 14}};
```

This statement declares `board` to be a two-dimensional array of `four` rows and `three` columns. The components of the first row are `2`, `3`, and `1`; the components of the second row are `15`, `25`, and `13`; the components of the third row are `20`, `4`, and `7`; and the components of the fourth row are `11`, `18`, and `14`, respectively. Figure 9-12 shows the array `board`.

board	[0]	[1]	[2]
[0]	2	3	1
[1]	15	25	13
[2]	20	4	7
[3]	11	18	14

FIGURE 9-12 Two-dimensional array `board`

To initialize a two-dimensional array when it is declared:

1. The elements of each row are enclosed within curly braces and separated by commas.
2. All rows are enclosed within curly braces.
3. For number arrays, if all components of a row are not specified, the unspecified components are initialized to `0`. In this case, at least one of the values must be given to initialize all the components of a row.

PROCESSING TWO-DIMENSIONAL ARRAYS

A two-dimensional array can be processed in three ways:

1. Process the entire array.
2. Process a particular row of the array, called **row processing**.
3. Process a particular column of the array, called **column processing**.

Initializing and printing the array are examples of processing the entire two-dimensional array. Finding the largest element in a row (column) or finding the sum of a row (column) are examples of row (column) processing. We will use the following declaration for our discussion:

```
const int NUMBER_OF_ROWS = 7;     //This can be set to any number.
const int NUMBER_OF_COLUMNS = 6; //This can be set to any number.

int matrix[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS];
int row;
int col;
int sum;
int largest;
int temp;
```

Figure 9-15 shows the array `matrix`.

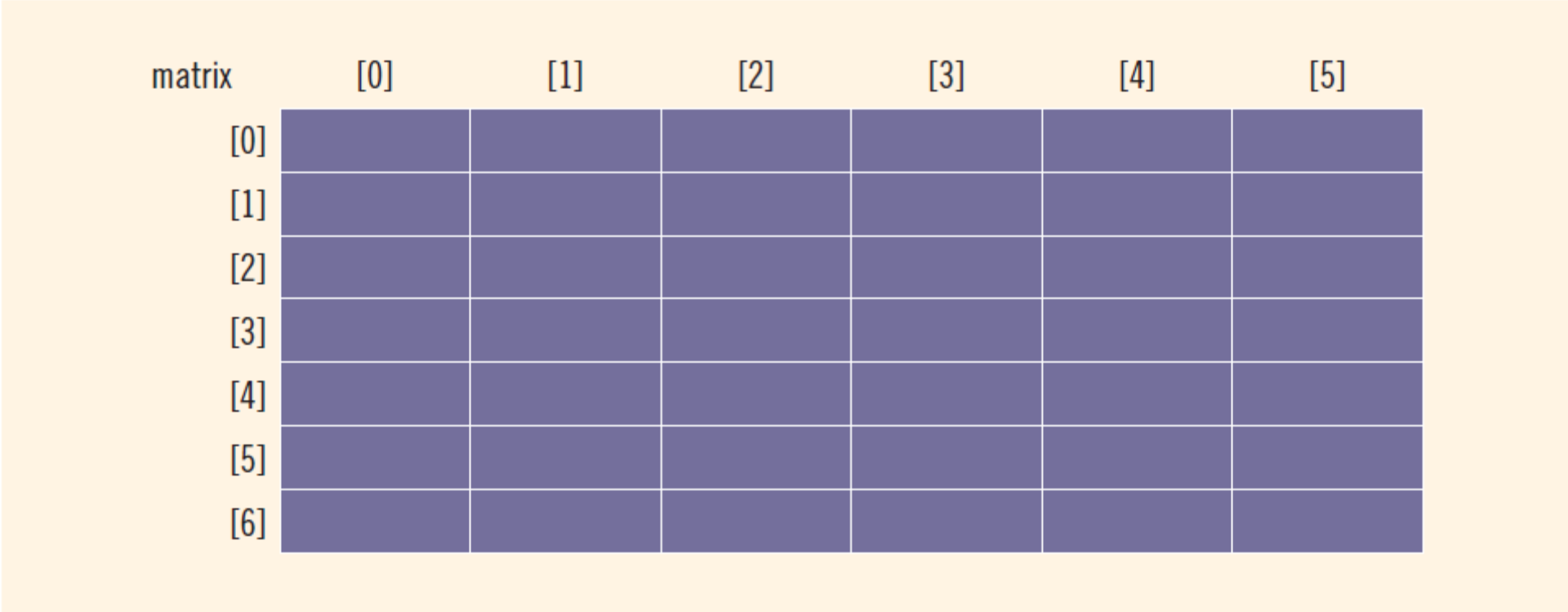


FIGURE 9-15 Two-dimensional array `matrix`

Because the components of a two-dimensional array are of the same type, the components of any row or column are of the same type. This means that each row and each column of a two-dimensional array is a one-dimensional array. Therefore, when processing a particular row or column of a two-dimensional array, we use algorithms similar to those that process one-dimensional arrays. We further explain this concept with the help of the two-dimensional array `matrix`, as declared previously.

Suppose that we want to process row number 5 of matrix (that is, the sixth row of matrix). The components of row number 5 of matrix are:

```
matrix[5][0], matrix[5][1], matrix[5][2], matrix[5][3], matrix[5][4],
matrix[5][5]
```

We see that in these components, the first index (the row position) is fixed at 5. The second index (the column position) ranges from 0 to 5. Therefore, we can use the following for loop to process row number 5:

```
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    process matrix[5][col]
```

Clearly, this for loop is equivalent to the following for loop:

```
row = 5;
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    process matrix[row][col]
```

Similarly, suppose that we want to process column number 2 of `matrix`, that is, the third column of `matrix`. The components of this column are:

```
matrix[0][2], matrix[1][2], matrix[2][2], matrix[3][2], matrix[4][2],
matrix[5][2], matrix[6][2]
```

Here, the second index (that is, the column position) is fixed at 2. The first index (that is, the row position) ranges from 0 to 6. In this case, we can use the following `for` loop to process column 2 of `matrix`:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
    process matrix[row][2]
```

Clearly, this `for` loop is equivalent to the following `for` loop:

```
col = 2;
for (row = 0; row < NUMBER_OF_ROWS; row++)
    process matrix[row][col]
```

Next, we discuss specific processing algorithms.

Initialization

Suppose that you want to initialize row number 4, that is, the fifth row, to 0. As explained earlier, the following `for` loop does this:

```
row = 4;
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    matrix[row][col] = 0;
```

If you want to initialize the entire `matrix` to 0, you can also put the first index, that is, the row position, in a loop. By using the following nested `for` loops, we can initialize each component of `matrix` to 0:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        matrix[row][col] = 0;
```

Print

By using a nested `for` loop, you can output the components of `matrix`. The following nested `for` loops print the components of `matrix`, one row per line:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
{
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        cout << setw(5) << matrix[row][col] << " ";

    cout << endl;
}
```

Input

The following `for` loop inputs the data into row number `4`, that is, the fifth row of `matrix`:

```
row = 4;

for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    cin >> matrix[row][col];
```

As before, by putting the row number in a loop, you can input data into each component of `matrix`. The following `for` loop inputs data into each component of `matrix`:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        cin >> matrix[row][col];
```

Sum by Row

The following `for` loop finds the sum of row number `4` of `matrix`; that is, it adds the components of row number `4`.

```
sum = 0;
row = 4;
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    sum = sum + matrix[row][col];
```

Once again, by putting the row number in a loop, we can find the sum of each row separately. Following is the C++ code to find the sum of each individual row:

```
  //Sum of each individual row
for (row = 0; row < NUMBER_OF_ROWS; row++)
{
    sum = 0;
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        sum = sum + matrix[row][col];

    cout << "Sum of row " << row + 1 << " = " << sum << endl;
}
```

Sum by Column

As in the case of sum by row, the following nested `for` loop finds the sum of each individual column:

```
  //Sum of each individual column
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
{
     sum = 0;
     for (row = 0; row < NUMBER_OF_ROWS; row++)
          sum = sum + matrix[row][col];

     cout << "Sum of column " << col + 1 << " = " << sum
          << endl;
}
```

Largest Element in Each Row and Each Column

As stated earlier, two other operations on a two-dimensional array are finding the largest element in each row and each column and finding the sum of both diagonals. Next, we give the C++ code to perform these operations.

The following `for` loop determines the largest element in row number `4`:

```
row = 4;
largest = matrix[row][0]; //Assume that the first element of
                          //the row is the largest.
for (col = 1; col < NUMBER_OF_COLUMNS; col++)
    if (largest < matrix[row][col])
        largest = matrix[row][col];
```

The following C++ code determines the largest element in each row and each column:

```
  //Largest element in each row
for (row = 0; row < NUMBER_OF_ROWS; row++)
{
    largest = matrix[row][0]; //Assume that the first element
                              //of the row is the largest.
    for (col = 1; col < NUMBER_OF_COLUMNS; col++)
        if (largest < matrix[row][col])
            largest = matrix[row][col];

    cout << "The largest element in row " << row + 1 << " = "
         << largest << endl;
}

  //Largest element in each column
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
{
    largest = matrix[0][col]; //Assume that the first element
                              //of the column is the largest.
    for (row = 1; row < NUMBER_OF_ROWS; row++)
        if (largest < matrix[row][col])
            largest = matrix[row][col];

    cout << "The largest element in column " << col + 1
         << " = " << largest << endl;
}
```

References

1. Malik, D. S. (2010). *C++ programming: Program design including data structures.* Course Technology.