# Lab Session 08

## 1-Dimensional and Multi-Dimensional Arrays in C++ Language.

**Objectives:**

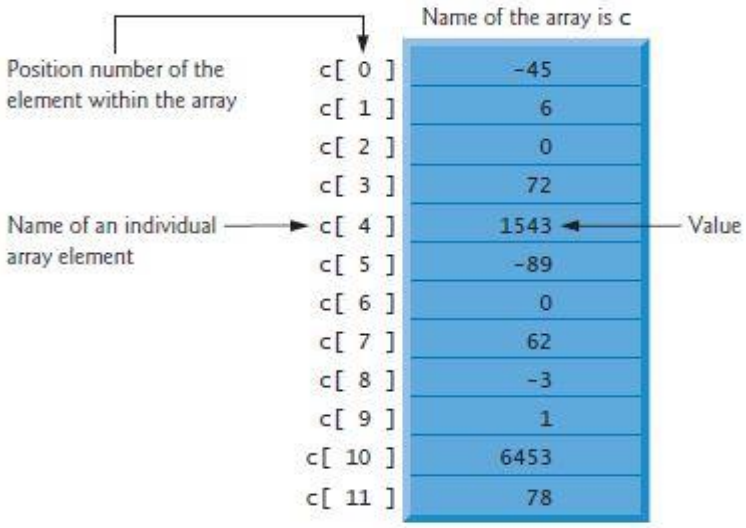
1. Illustration of Arrays.
2. To learn the syntax and semantics of the 1-D and Multi-Dimensional Arrays in C++ programming language.
3. Demonstrate a thorough understanding of Arrays by designing and implementing programs logic.

**Arrays:**

An array is a consecutive group of memory locations that all have the same type. To refer to a particular location or element in the array, we specify the name of the array and the position number of the particular element in the array.

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

In books on C programming, structures are often considered an advanced feature and are introduced toward the end of the book. However, for C++ programmers, structures are one of the two important building blocks in the understanding of objects and classes. In fact, the syntax of a structure is almost identical to that of a class. A structure (as typically used) is a collection of data, while a class is a collection of both data and functions. So by learning about structures we'll be paving the way for an understanding of classes and objects. Structures in C++ (and C) serve a similar purpose to *records* in some other languages such as Pascal.




A subscript must be an integer or integer expression (using any integral type). If a

program uses an expression as a subscript, then the program evaluates the expression to determine the subscript. For example, if we assume that variable a is equal to 5 and that variable b is equal to 6, then the statement adds 2 to array element c[11]. A subscripted array name is an *lvalue*—it can be used on the left side of an assignment, just as nonarray variable names can.

```
c[ a + b ] += 2;
```

Let's examine array c in Fig. 1 more closely. The name of the entire array is c. Its 12 elements are referred to as c[0] to c[11]. The value of c[0] is -45, the value of c[1] is 6, the value of c[2] is 0, the value of c[7] is 62, and the value of c[11] is 78. To print the sum of the values contained in the first three elements of array c, we'd write:

```
cout << c[ 0 ] + c[ 1 ] + c[ 2 ] << endl;
```

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

```
x = c[ 6 ] / 2;
```

The brackets that enclose a subscript are actually an operator that has the same precedence as parentheses. Figure 2 shows the precedence and associativity of the operators introduced so far. The operators are shown top to bottom in decreasing order of precedence with their associativity and type.

| Operators | Associativity | Type |
|---|---|---|
| :: () | *[See parentheses caution in Fig. 2.10]* | scope resolution |
| () [] | left to right | function call/array access |
| ++ -- static_cast<*type*>(*operand*) | left to right | unary (postfix) |
| ++ -- + - ! | right to left | unary (prefix) |
| * / % | left to right | multiplicative |
| + - | left to right | additive |
| << >> | left to right | insertion/extraction |
| < <= > >= | left to right | relational |
| == != | left to right | equality |
| && | left to right | logical AND |
| \|\| | left to right | logical OR |
| ?: | right to left | conditional |
| = += -= *= /= %= | right to left | assignment |
| , | left to right | comma |

Operator precedence and associativity

**Declaring Arrays:**

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

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

The compiler reserves the appropriate amount of memory. (Recall that a declaration which reserves memory is more properly known as a definition.) The arraySize must be an integer

constant greater than zero. For example, to tell the compiler to reserve 12 elements for integer array c, use the declaration. Arrays can be declared to contain values of any nonreference data type. For example, an array of type string can be used to store character strings.

```
int c[ 12 ]; // c is an array of 12 integers
```

**Example 01:**

This program, REPLAY, creates an array of four integers representing the ages of four people. It then asks the user to enter four values, which it places in the array. Finally, it displays all four values.

```
#include <iostream>
using namespace std;
int main()
{
        int age[4]; //array 'age' of 4 ints
for(int j=0; j<4; j++) //get 4 ages
        {
        cout << "Enter an age: ";
        cin >> age[j]; //access array element
        }
        for(j=0; j<4; j++) //display 4 ages
        cout << "You entered " << age[j] << endl;
        return 0;
}
```

Here's a sample interaction with the program:

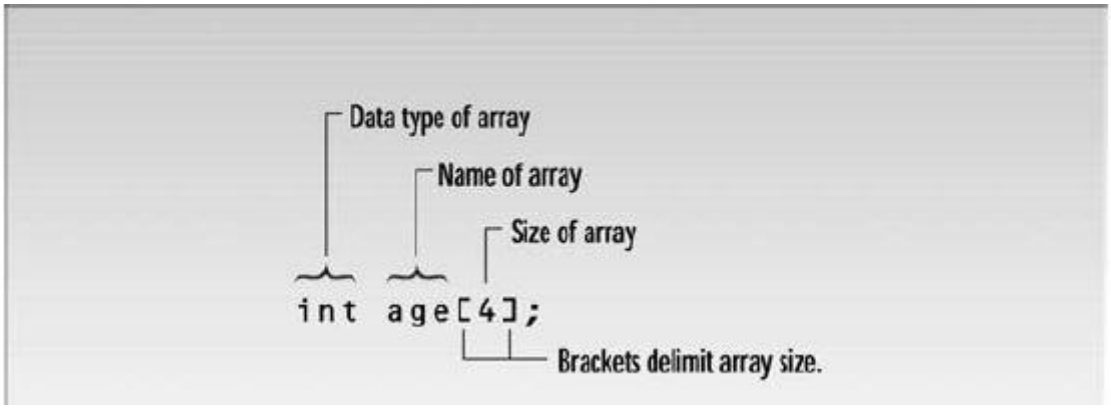
```
Enter an age: 44
Enter an age: 16
Enter an age: 23
Enter an age: 68
You entered 44
You entered 16
You entered 23
You entered 68
```

The first for loop gets the ages from the user and places them in the array, while the second reads them from the array and displays them.

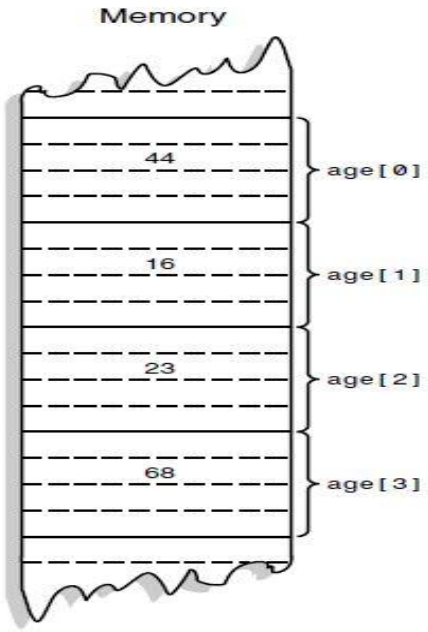
In the REPLAY example, the array is type int. The name of the array comes next, followed immediately by an opening bracket, the array size, and a closing bracket. The number in brackets must be a constant or an expression that evaluates to a constant, and should also be an integer. In the example we use the value 4.

**Array Elements:**

The items in an array are called *elements* (in contrast to the items in a structure, which are called *members*). As we noted, all the elements in an array are of the same type; only the values vary. Figure 3 shows the elements of the array age.




Syntax of Array definition




Array Elements

Following the conventional (although in some ways backward) approach, memory grows downward in the figure. That is, the first array elements are on the top of the page; later elements extend downward. As specified in the definition, the array has exactly four elements.

Notice that the first array element is numbered 0. Thus, since there are four elements, the last one is number 3. This is a potentially confusing situation; you might think the last element in a four-element array would be number 4, but it's not.

**Accessing Array Elements:**

In the REPLAY example we access each array element twice. The first time, we insert a value into the array, with the line

cin >> age[j];

The second time, we read it out with the line

cout << "\nYou entered " << age[j];

In both cases the expression for the array element is

age[j]

This consists of the name of the array, followed by brackets delimiting a variable j. Which of the four array elements is specified by this expression depends on the value of j; age[0] refers to the first element, age[1] to the second, age[2] to the third, and age[3] to the fourth. The variable (or constant) in the brackets is called the *array index*.

Since j is the loop variable in both for loops, it starts at 0 and is incremented until it reaches 3, thereby accessing each of the array elements in turn.

**Averaging Array Elements**

Here's another example of an array at work. This one, SALES, invites the user to enter a series of six values representing widget sales for each day of the week (excluding Sunday), and then calculates the average of these values. We use an array of type double so that monetary values can be entered.

```
#include <iostream>
using namespace std;
int main()
{
        const int SIZE = 6; //size of array
        double sales[SIZE]; //array of 6 variables
        cout << "Enter widget sales for 6 days\n";
        for(int j=0; j<SIZE; j++) //put figures in array
                cin >> sales[j];
        double total = 0;
        for(j=0; j<SIZE; j++) //read figures from array
        total += sales[j]; //to find total
        double average = total / SIZE; // find average
        cout << "Average = " << average << endl;
        return 0;
}
```

Here's some sample interaction with SALES:

```
Enter widget sales for 6 days
        352.64
        867.70
        781.32
        867.35
        746.21
        189.45
Average = 634.11
```

A new detail in this program is the use of a const variable for the array size and loop limits. This variable is defined at the start of the listing:

```
const int SIZE = 6;
```

Using a variable (instead of a number, such as the 4 used in the last example) makes it easier to change the array size: Only one program line needs to be changed to change the array size, loop limits, and anywhere else the array size appears. The all-uppercase name reminds us that the variable cannot be modified in the program.

**Initializing Arrays:**

You can give values to each array element when the array is first defined. Here's an example, DAYS, that sets 12 array elements in the array days_per_month to the number of days in each month.

```
#include <iostream>
using namespace std;
int main()
        {
        int month, day, total_days;
        int days_per_month[12] = { 31, 28, 31, 30, 31, 30,
        31, 31, 30, 31, 30, 31 };
        cout << "\nEnter month (1 to 12): "; //get date
        cin >> month;
        cout << "Enter day (1 to 31): ";
        cin >> day;
        total_days = day; //separate days
        for(int j=0; j<month-1; j++) //add days each month
        total_days += days_per_month[j];
        cout << "Total days from start of year is: " << total_days
```

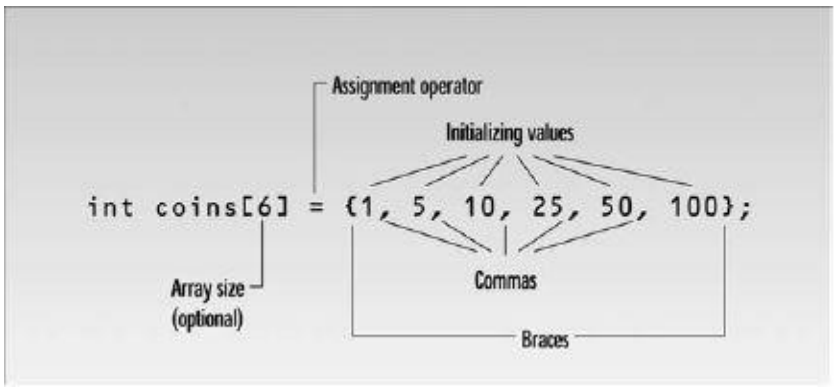
```
<< endl;
return 0;
}
```

The program calculates the number of days from the beginning of the year to a date specified by the user. (Beware: It doesn't work for leap years.) Here's some sample interaction:

```
Enter month (1 to 12): 3
Enter day (1 to 31): 11
Total days from start of year is: 70
```

Once it gets the month and day values, the program first assigns the day value to the total_days variable. Then it cycles through a loop, where it adds values from the days_per_month array to total_days. The number of such values to add is one less than the number of months. For instance, if the user enters month 5, the values of the first four array elements (31, 28, 31, and 30) are added to the total.

The values to which days_per_month is initialized are surrounded by braces and separated by commas. They are connected to the array expression by an equal sign. Figure 4 shows the syntax.




Array Initialization

Actually, we don't need to use the array size when we initialize all the array elements, since the compiler can figure it out by counting the initializing variables. Thus we can write

```
int days_per_month[] = { 31, 28, 31, 30, 31, 30,
                         31, 31, 30, 31, 30, 31 };
```

What happens if you do use an explicit array size, but it doesn't agree with the number of initializers? If there are too few initializers, the missing elements will be set to 0. If there are too many, an error is signaled.

**Example 02:**

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

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
        int n[ 10 ]; // n is an array of 10 integers
        // initialize elements of array n to 0
        for ( int i = 0; i < 10; ++i )
        n[ i ] = 0; // set element at location i to 0

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

        // output each array element's value
        for ( int j = 0; j < 10; ++j )
        cout << setw( 7 ) << j << setw( 13 ) << n[ j ] << endl;
} // end main
```

**Output:**

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

**Example 03:**

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

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

int main()
{
        // use initializer list to initialize array n
        int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
        cout << "Element" << setw( 13 ) << "Value" << endl;
        // output each array element's value
        for ( int i = 0; i < 10; ++i )
                cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;

} // end main
```

**Output:**

```
Element       Value
0             32
1             27
2             64
3             18
4              95
5             14
6             90
7             70
8             60
9             37
```

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

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

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

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

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

If the array size and an initializer list are specified in an array declaration, the number of initializers must be less than or equal to the array size. The array declaration causes a compilation error, because there are six initializers and only five array elements.

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

**Example 04:**
Following program sets the elements of a 10-element array s to the even integers 2, 4, 6, …, 20 (lines 14–15) and prints the array in tabular format (lines 17–21). These numbers are generated (line 15) by multiplying each successive value of the loop counter by 2 and adding 2.

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

int main()
{

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

    for ( int i = 0; i < arraySize; ++i ) // set the values
    s[ i ] = 2 + 2 * i;
    cout << "Element" << setw( 13 ) << "Value" << endl;
    // output contents of array s in tabular format
    for ( int j = 0; j < arraySize; ++j )
        cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << endl;
} // end main
```

**Output ???**

**Example 05:**

**Summing the Elements of an Array**

```
#include <iostream>
using namespace std;
int main()
{
	const int arraySize = 10; // constant variable indicating size of array
	int a[ arraySize ] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
	int total = 0;
	// sum contents of array a
	for ( int i = 0; i < arraySize; ++i )
	total += a[ i ];

	cout << "Total of array elements: " << total << endl;
} // end main
```

**Output:**
Total of array elements: 849

**Multidimensional Arrays**

So far we've looked at arrays of one dimension: A single variable specifies each array element. But arrays can have higher dimensions. Here's a program, SALEMON that uses a two-dimensional array to store sales figures for several districts and several months:
**Definition**

The array is defined with two size specifiers, each enclosed in brackets:

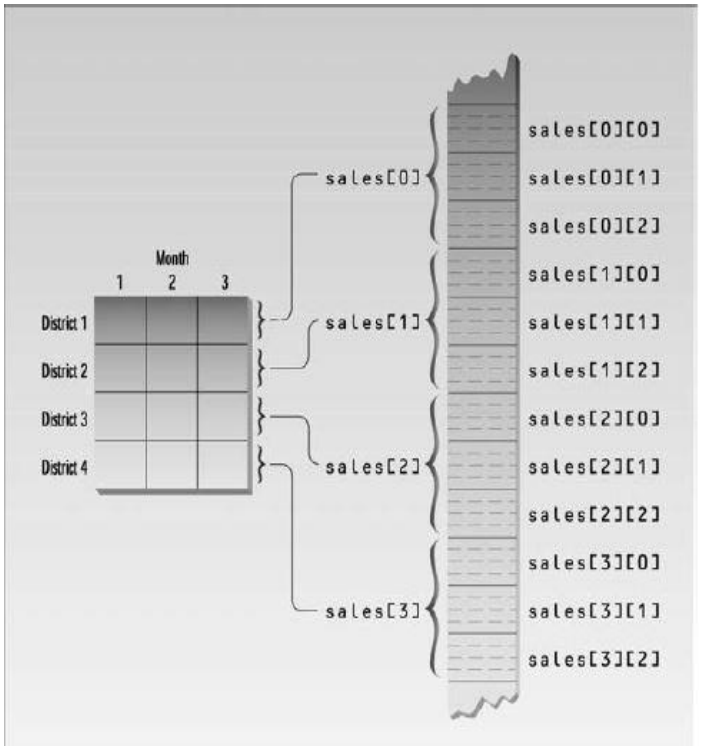
```
	double sales[DISTRICTS][MONTHS];
```

You can think about sales as a two-dimensional array, laid out like a checkerboard. Anotherway to think about it is that sales is an array of arrays. It is an array of DISTRICTS elements, each of which is an array of MONTHS elements. Figure 4 shows how this looks.

Of course there can be arrays of more than two dimensions. A three-dimensional array is an array of arrays of arrays. It is accessed with three indexes:

```
elem = dimen3[x][y][z];
```

This is entirely analogous to one- and two-dimensional arrays.




**Example 06:**

```
#include <iostream>
#include <iomanip> //for setprecision, etc.
using namespace std;
const int DISTRICTS = 4; //array dimensions
const int MONTHS = 3;
int main()
        {
        int d, m;
        double sales[DISTRICTS][MONTHS]; //two-dimensional array definition
        cout << endl;
        for(d=0; d<DISTRICTS; d++) //get array values
        for(m=0; m<MONTHS; m++)
        {
        cout << "Enter sales for district " << d+1;
        cout << ", month " << m+1 << ": ";
        cin >> sales[d][m]; //put number in array

cout << "\n\n";
cout << " Month\n";
cout << " 1 2 3";
for(d=0; d<DISTRICTS; d++)
        {
        cout <<"\nDistrict " << d+1;
```

```
        for(m=0; m<MONTHS; m++) //display array values
        cout << setiosflags(ios::fixed) //not exponential
                << setiosflags(ios::showpoint) //always use point
                << setprecision(2) //digits to right
                << setw(10) //field width
                << sales[d][m]; //get number from array
        } //end for(d)

cout << endl;
return 0;
} //end main
```

This program accepts the sales figures from the user and then displays them in a table.
Enter sales for district 1, month 1: 3964.23
Enter sales for district 1, month 2: 4135.87
Enter sales for district 1, month 3: 4397.98
Enter sales for district 2, month 1: 867.75
Enter sales for district 2, month 2: 923.59
Enter sales for district 2, month 3: 1037.01
Enter sales for district 3, month 1: 12.77
Enter sales for district 3, month 2: 378.32
Enter sales for district 3, month 3: 798.22
Enter sales for district 4, month 1: 2983.53
Enter sales for district 4, month 2: 3983.73
Enter sales for district 4, month 3: 9494.98

Output:

**Tasks/Assignment:**

1. **Write source code of C++ program to add two m x n matrix.**
2. **Write source code of C++ program to subtract two m x n matrix.**
3. **Write source code of C++ Program to Multiply Two Matrices.**