Computing 4

C++ PROGRAMMING

BY: SAAD AL-MOMEN

Department of Mathematics | 2nd Class | 2nd Semester
College of Science | University of Baghdad

1

Arrays

1. Introduction

Let's write a program to read five numbers, find their sum, and print the numbers in reverse order.

```
#include <iostream>
using namespace std;
int main()
{
        int item0, item1, item2, item3, item4;
        int sum;

        cout << "Enter five integers: ";
        cin >> item0 >> item1 >> item2 >> item3 >> item4;
        cout << endl;

        sum = item0 + item1 + item2 + item3 + item4;

        cout << "The sum of the numbers = " << sum << endl;
        cout << "The numbers in the reverse order are: ";
        cout << item4 << " " << item3 << " " << item2 << " "<< item1 << " " << item0<<
        endl;
        return 0;
}
```

In the previous program:

- Five variables were declared.
- All variables are of type **int**, i.e. they are of the same data type.
- The way in which these variables are declared indicates that the variables to store these numbers all have the same name—except the last character, which is a number.

This program works fine. However, if you need to read 100 (or more) numbers and print them in reverse order, you would have to declare 100 variables and write many **cin** and **cout** statements. Thus, for large amounts of data, this type of program is not desirable.

Because all variables are of the same type, you should be able to specify how many variables must be declared—and their data type—with a simpler statement than the one we used earlier.

The data structure that lets you do all of these things in C++ is called an **array**.

2. Arrays in One Dimension

An **array** is a collection of a fixed number of components all of the same data type. A **one-dimensional array** is an array in which the components are arranged in a list form. The general form for declaring a one-dimensional array is:

```
dataType   arrayName [arraySize];
```

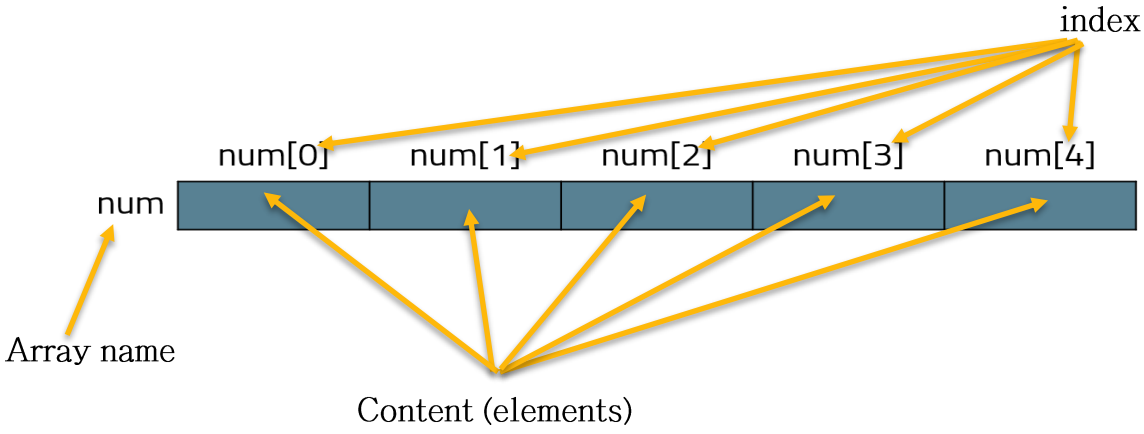
Example 1.1:

The statement:

```
int num[5];
```

declares an array **num** of five components (elements). Each element is of type **int**.

The elements are **num[0]**, **num[1]**, **num[2]**, **num[3]**, and **num[4]**.



Example 1.2:

Here are some other examples of declaring one dimensional arrays

```
int age [10];
int num [5];
float temp [365];
char alphabet [26];
```

3. Accessing Array Components

The general form (syntax) used for accessing an array component is:

```
arrayName [indexExp]
```

in which indexExp, called the **index**, is any expression whose value is a nonnegative integer. The index value specifies the position of the component in the array.

In C++, [ ] is an operator called the array subscripting operator. Moreover, in C++, the array index starts at 0.

4. Initializing Arrays

You can initialize C++ array elements either one by one or using a single statement as follows:

```
float balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

```
float balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

You will create exactly the same array as you did in the previous example.

```
balance[4] = 50.0;
```

The above statement assigns element number 5th in the array a value of 50.0. Array with 4^{th} index will be 5^{th} , i.e., last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representation of the same array we discussed above:

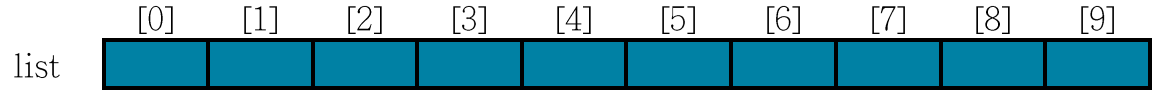
	0	1	2	3	4
balance	1000.0	2.0	3.4	17.0	50.0

5. Accessing Array Elements

Consider the following statement:

```
int list[10];
```

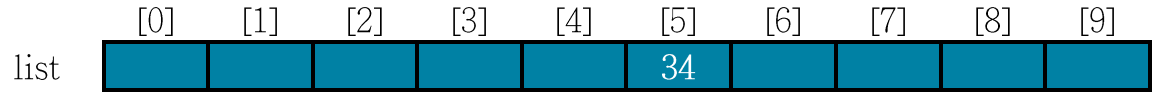
This statement declares an array list of 10 components. The components are list[0], list[1], . . ., list[9]. In other words, we have declared 10 variables



The assignment statement:

```
list[5] = 34;
```

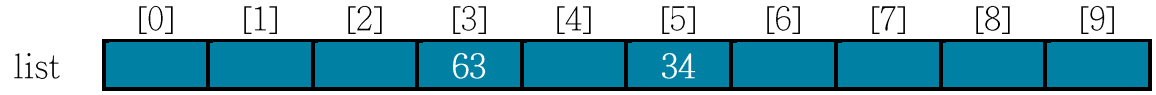
stores 34 in list[5], which is the sixth component of the array list.



Suppose i is an int variable. Then, the assignment statement:

```
i = 3;
list[i] = 63;
```

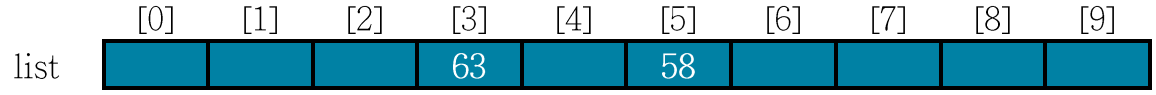
gives:



If i is 4, then the assignment statement:

```
list[2 * i - 3] = 58;
```

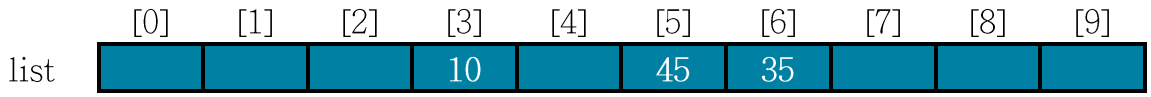
stores 58 in list[5] because 2 * i - 3 evaluates to 5. The index expression is evaluated first, giving the position of the component in the array.



Next, consider the following statements:

```
list[3] = 10;
list[6] = 35;
list[5] = list[3] + list[6];
```

The first statement stores 10 in list[3], the second statement stores 35 in list[6], and the third statement adds the contents of list[3] and list[6] and stores the result in list[5].



6. Processing One-Dimensional Arrays

suppose that we have the following statements:

```
int list[100]; //list is an array of size 100
int i;
```

The following for loop steps through each element of the array list, starting at the first element of list:

```
for (i = 0; i < 100; i++)        //Line 1
//process list[i]                //Line 2
```

If processing the list requires inputting data into list, the statement in Line 2 takes the form of an input statement, such as the cin statement. For example, the following statements read 100 numbers from the keyboard and store the numbers in list:

```
for (i = 0; i < 100; i++)        //Line 1
cin >> list[i];                  //Line 2
```

Similarly, if processing list requires outputting the data, then the statement in Line2 takes the form of an output statement.

Example 1.3:

Write a program to read five numbers, find their sum, and print the numbers in reverse order.

```
#include <iostream>
using namespace std;
int main()
{
        int item[5]; //Declare an array item of five components
        int sum;
```

```
        int counter;
        cout << "Enter five numbers: ";
        sum = 0;
        for (counter = 0; counter < 5; counter++)
        {
                cin >> item[counter];
                sum = sum + item[counter];
        }
        cout << endl;
        cout << "The sum of the numbers is: " << sum << endl;
        cout << "The numbers in reverse order are: ";
        //Print the numbers in reverse order.
        for (counter = 4; counter >= 0; counter--)
                cout << item[counter] << " ";
        cout << endl;
return 0;
}
```

Sample Run: In this sample run, the user input is shaded.
Enter five numbers: 12 76 34 52 89
The sum of the numbers is: 263
The numbers in reverse order are: 89 52 34 76 12

Example 1.4:

Write a program to display 2nd and 4th elements of array vect:

```
#include <iostream>
using namespace std;
int main()
{
        float vect[ ] = { 1.7, 7.23, 98.01, 4.98, 2.88};
        cout << "2nd element is: " << vect[1] << endl;
```

```
        cout << "4th element is: "<< vect[3];
        return 0;
}
```

Example 1.5:

Write a program to find the summation of array elements

```
#include <iostream>
using namespace std;
int main() {
   int arr[] = {1, 2, 3, 4, 5};
   int size = 5;
   // Calculating the sum
   int sum = 0;
   for (int i = 0; i < size; i++) {
      sum += arr[i];
   }
   // Displaying the result
   cout << "Sum of array elements: " << sum << endl;
   return 0;
}
```

Example 1.6:

Rewrite example 1.5 to give the user the ability to enter the elements of the array.

```
#include <iostream>
using namespace std;
int main() {
   const int L = 5;
   int a[L];
   int sum = 0;
```

```
    cout << "Enter 5 numbers:\n";
    for (int i = 0; i < L; i++) {
        cout << "Enter value " << i << ": ";
        cin >> a[i];
        sum =sum + a[i];
    }
    cout << "Sum is: " << sum << endl;
    return 0;
}
```

Example 1.7:

Write a program to find the minimum value in an array

```
#include <iostream>
using namespace std;
int main() {
    const int size = 8;
    int numbers[size];
    // Input
    cout << "Enter 8 numbers:\n";
    for (int i = 0; i < size; i++) {
        cout << "Enter number " << i+1 << ": ";
        cin >> numbers[i];
    }
    // Finding minimum
    int min = numbers[0]; // Assume the first number is the minimum
    for (int i = 1; i < size; i++) {
        if (numbers[i] < min) {
            min = numbers[i]; // Update min if a smaller number is found
        }
```

```
    }


    // Output
    cout << "Minimum value is: " << min << endl;
    return 0;
}
```

Example 1.8:

Create a program that separates odd and even numbers from a given array into two distinct arrays.

```
#include <iostream>
using namespace std;
int main() {
    int a[10] = { 14, 32, 13, 41, 52, 61, 71, 83, 2, 7};
    int oddv[10], evenv[10];
    int i, n1 = 0, n2 = 0;

    for (i = 0; i < 10; i++) {
        if (a[i] % 2 != 0) {
            oddv[n1] = a[i];
            n1= n1 +1;
        } else {
            evenv[n2] = a[i];
            n2=n2+1;
        }
    }
    cout << "Odd numbers: ";
    for (i = 0; i < n1; i++) {
        cout << oddv[i] << " ";
```

```
    }
    cout << endl << "Even numbers: ";
    for (i = 0; i < n2; i++) {
        cout << evenv[i] << " ";
    }
    return 0;
}
```

Example 1.9:

Write a program to read an array of 30 real numbers, the program calculates the sum and the average and find the maximum element of the array.

```
#include<iostream>
using namespace std;
int main() {
    float A[10];
    int i;
    for (i = 0; i < 10; i++)
        cin >> A[i];
    float sum = 0; // Initialize sum
    float max = A[0];
    for (i = 0; i < 10; i++) {
        sum = sum + A[i];
        if (A[i] > max)
            max = A[i];
    }
    float av = sum / 10; // Calculate average
    cout << "Sum: " << sum << ", Average: " << av << ", Maximum: " << max << endl;
    return 0;
}
```

Example 1.10:

Write a program to sort an array of 10 integers in an ascending order.

```
#include <iostream>
using namespace std;
int main() {
   int v[10] = {5, 2, 9, 1, 6, 3, 7, 8, 4, 10};
   int n = 10;
    for (int i = 0; i < n - 1; i++) {
      for (int j = i+1; j <= n -1; j++) {
         if (v[i] > v[j]) {
            int t = v[i];
            v[i] = v[j];
            v[j ] = t;
         }
      }
   }
   cout << "Sorted array in ascending order: ";
   for (int i = 0; i < n; i++) {
      cout << v[i] << " ";
   }
   cout << endl;
   return 0;
}
```

Example 1.11:

Write C++ program to initialize array of even integers from 2 to 20.

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

```
int main()
        {
        const int arraysize=10;
        int j,s[arraysize];
        for(j=0;j<arraysize;j++)
        s[j]=2+2*j;
        cout<<"Element"<<setw(23)<<"Initialized value"<<endl;
        for(j=0;j<10;j++)
        cout<<setw(7)<<j<<setw(13)<<s[j]<<endl;
return 0;
        }
```

The output for the above program will be as follows:

```
Element     Initialized value
    0           2
    1           4
    2           6
    3           8
    4          10
    5          12
    6          14
    7          16
    8          18
    9          20
```

Example 1.12:

Write C++ program to initialize array s[] to the odd integers from 1 to 40 and it's inverse.

```
#include <iostream>
using namespace std;
int main()
        {
```

```
const int arraysize=20;
int s[arraysize];
for(int i=1;i<21;i++)
        {
        s[i]=2*i-1;
        cout<<s[i]<<'\t';
        }
        cout<<endl;
        cout<<"The inverse values of the above array are"<<endl;
        //inverse the array values
        for(int z=20;z>0;z--)
        cout<<s[z]<<"\t";
   return 0;
}
```

The output for the above program will be as follows:

```
1       3       5       7       9       11      13      15      17      19      21      23
        25      27      29      31      33      35      37      39

The inverse values of the above array are
39      37      35      33      31      29      27      25      23      21      19      17
        15      13      11      9       7       5       3       1
```

7. Get the Size of an Array

To get the size of an array, you can use the **sizeof()** operator:

```
int myNumbers[5] = {10, 20, 30, 40, 50};
cout << sizeof(myNumbers);
```

The result for the above program will be 20!.
Why did the result show 20 instead of 5, when the array contains 5 elements? It is because the **sizeof()** operator returns the size of a type in **bytes**. An **int** type is usually 4 bytes, so from the example above, *4 x 5 (4 bytes x 5 elements)* = **20 bytes**.

To find out how many elements an array has, you have to divide the size of the array by the size of the data type it contains:

```
int myNumbers[5] = {10, 20, 30, 40, 50};
int getArrayLength = sizeof(myNumbers) / sizeof(int);
cout << getArrayLength;
```

In this case the result will be 5.

8. Loop Through an Array with sizeof()

Previously, we wrote the size of the array in the loop condition (i < 5). This is not ideal, since it will only work for arrays of a specified size.
However, by using the **sizeof()** approach from the example above, we can now make loops that work for arrays of any size, which is more sustainable.

Instead of writing:

```
int myNumbers[5] = {10, 20, 30, 40, 50};
for (int i = 0; i < 5; i++) {
  cout << myNumbers[i] << "\n";
}
```

It is better to write:

```
int myNumbers[5] = {10, 20, 30, 40, 50};
for (int i = 0; i < sizeof(myNumbers) / sizeof(int); i++) {
  cout << myNumbers[i] << "\n";
}
```

Note that, in C++ version 11 (2011), you can also use the "for-each" loop, the syntax for this command is:

```
for (type variableName : arrayName) {
  // code block to be executed
}
```

for example, the above code can be replaced by:

```
int myNumbers[5] = {10, 20, 30, 40, 50};
for (int i : myNumbers) {
  cout << i << "\n";
}
```

Exercisers

1. Write a program to find the average of even and odd numbers in an array of 10 numbers.
2. Write a program to sort an array of 10 integers in a descending order.
3. Write a program that searches for a given number in an integer array and prints its index if found
4. Write a program that copies the elements of one integer array to another and prints the copied array
5. Write a program that adds two arrays of the same size element-wise and stores the result in a third array

2

Two Dimensional Arrays

1. Introduction

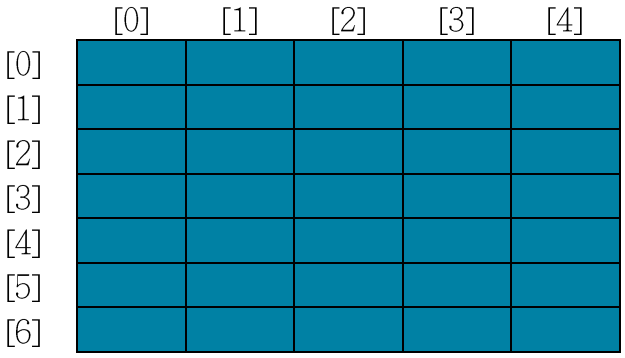
A **two-dimensional array** is a collection of a fixed number of components arranged in rows and columns (that is, in two dimensions), wherein all components are of the same type. The syntax for declaring a two-dimensional array is:

```
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[7][5];
```

declares a two-dimensional array sales of 7 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. . .6 and the columns are numbered 0. . .4.



2. Accessing Array Components

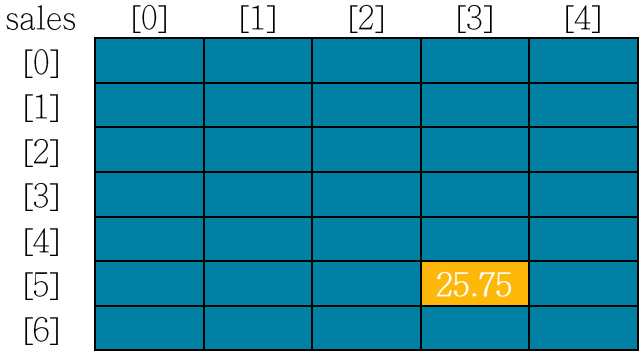
To access the components of a two-dimensional array, you need a pair of indices: one for the row position and one for the column position. The syntax to access a component of a two-dimensional array is:

```
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



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.

3. 2D Array Initialization During Declaration

Like one-dimensional arrays, two-dimensional arrays can be initialized when they are declared. Consider the following statement:

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

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

4. 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:

```
i = 4;
for (j = 0; j < cols; j++)
        matrix[i][j] = 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 (i = 0; i<rows; i++)
        for (j = 0; j< cols; j++)
           matrix[i][j] = 0;
```

5. 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 (i = 0; i < rows; i++)
{
for (j = 0; j < cols; j++)
        cout << setw(5) << matrix[i][j] << " "; cout << endl;
}
```

6. Input

The following for-loop inputs data into each component of matrix:

```
for (i = 0; i < rows; i++)
        for (j = 0; j < cols; j++)
            cin >> matrix[i][j];
```

Example 2.1:

Write a program, to read 15 numbers, 5 numbers per row, then print them:

```
#include<iostream>
using namespace std;
int main ( ) {
int a[3][5];
int i,j;
```

```
//Reading Matrix
for (i=0;i<3;i++)
  for (j=0;j<5;j++)
       cin >> a [i] [j] ;
//Writing Matrix
for (i=0;i<3;i++)
{
  for (j= 0;j< 5;j++)
      cout << a[i][j]<<"   ";
 cout<<endl;
}
return 0;
}
```

Example 2.2:

Write a program, to read 3×3 2D-array, then find the summation of the array elements, and finally print these elements

```
#include <iostream>
using namespace std;
int main() {
   const int rows = 3;
   const int cols = 3;
   int a[rows][cols];
   // Read the elements of the array
   cout << "Enter the elements of the 3x3 array:" << endl;
   for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         cin >> a[i][j];
      }
   }
   // Calculate the summation of the elements
   int sum = 0;
   for (int i = 0; i < rows; i++) {
```

```
        for (int j = 0; j < cols; j++) {
            sum += a[i][j];
        }
    }
    // Print the elements of the array
    cout << "Elements of the 3x3 array:" << endl;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cout << a[i][j] << " ";
        }
        cout << endl;
    }
    // Print the summation of the elements
    cout << "Summation of the array elements: " << sum << endl;
    return 0;
}
```

7. Sum by Row

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

```
sum = 0;
i = 4;
for (j = 0; j < cols; j++)
sum = sum + matrix[i][j];
```

Example 2.3:

Write a program, to read 3×4 2D-array, then find the summation of each row.

```
#include <iostream>
using namespace std;
int main() {
    const int rows = 3;
    const int cols = 4;
    int a[rows][cols];
    // Read the elements of the array
```

```
    cout << "Enter the elements of the 3x4 array:" << endl;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cin >> a[i][j];
        }
    }
    // Calculate the summation of each row
    int rowSum[rows] = {0}; // Initialize rowSum array with zeros
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            rowSum[i] += a[i][j];
        }
    }
    // Print the summation of each row
    cout << "Summation of each row:" << endl;
    for (int i = 0; i < rows; i++) {
        cout << "Row " << i + 1 << ": " << rowSum[i] << endl;
    }
    return 0;
}
```

8. Sum by Column

The following for loop finds the sum of column number 2 of a matrix; that is, it adds the components of column number 2.

```
sum = 0;
j = 2;
for (i= 0; i < rows; j++)
sum = sum + matrix[i][j];
```

Example 2.4:

Write a program, to read 3×4 2D-array, then find the summation of each column.

```
#include <iostream>
using namespace std;
int main() {
```

```
    const int rows = 3;
    const int cols = 4;
    int a[rows][cols];
    // Read the elements of the array
    cout << "Enter the elements of the 3x4 array:" << endl;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cin >> a[i][j];
        }
    }
    // Calculate the summation of each column
    int colSum[cols] = {0}; // Initialize colSum array with zeros
    for (int j = 0; j < cols; j++) {
        for (int i = 0; i < rows; i++) {
            colSum[j] += a[i][j];
        }
    }
    // Print the summation of each column
    cout << "Summation of each column:" << endl;
    for (int j = 0; j < cols; j++) {
        cout << "Column " << j + 1 << ": " << colSum[j] << endl;
    }
    return 0;
}
```

8. Largest Element in Each Row and Each Column

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

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

```
largest = matrix[i][j];
```

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

```
//Largest element in each row
for (i = 0; i < rows; i++)
{
largest = matrix[i][0]; //Assume that the first element of the row is the largest.
for (j = 1; j < cols; j++)
if (largest < matrix[i][j])
largest = matrix[i][j];
cout << "The largest element in row " << i + 1 << " = "<< largest << endl;
}
```

and the following code determines the largest element in each column:

```
//Largest element in each column
for (j = 0; j < cols; j++)
{
largest = matrix[0][j]; //Assume that the first element of the column is the largest.
for (i = 1; i < rows; i++)
if (largest < matrix[i][j])
largest = matrix[i][j];
cout << "The largest element in column " << j + 1 << " = " << largest << endl;
}
```

9. Fundamental Operations on Matrices (2D Arrays)

Operations on matrices include addition, subtraction, multiplication, and transposition. Addition and subtraction involve corresponding elements of matrices, while multiplication combines rows and columns based on specific rules. Transposition swaps rows with columns, serving essential roles in various mathematical, scientific, and engineering applications.

Example 2.5:

Write a program to read two 3x3 matrices (A and B), and compute their sum.

```
#include <iostream>
using namespace std;
int main() {
```

```
    const int rows = 3;
    const int cols = 3;
    int A[rows][cols];
    int B[rows][cols];
    int sum[rows][cols];
    // Read matrix A from the user
    cout << "Enter the elements of matrix A (3x3):" << endl;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cin >> A[i][j];       }
    }
    // Read matrix B from the user
    cout << "Enter the elements of matrix B (3x3):" << endl;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cin >> B[i][j];       }
    }
    // Compute the sum of matrices A and B
    cout << "Sum of matrices A and B:" << endl;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            sum[i][j] = A[i][j] + B[i][j];
            cout << sum[i][j] << " ";       }
        cout << endl;
    }
    return 0;
}
```

Example 2.6:

Write a program to find the product of two matrices A[n][m] and B[p][r].

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

```
int main() {
    int n, m, p, r;
    // Input the dimensions of matrices
    cout << "Enter the dimensions of matrix A (n x m): ";
    cin >> n >> m;
    cout << "Enter the dimensions of matrix B (p x r): ";
    cin >> p >> r;
    // Check if matrix multiplication is possible
    if (m != p) {
        cout << "Matrix multiplication is not possible. Please enter valid dimensions." << endl;
        return 0;
    }
    // Initialize matrices A and B
    int A[n][m], B[p][r];
    // Input matrix A
    cout << "Enter the elements of matrix A:" << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> A[i][j];
        }
    }
    // Input matrix B
    cout << "Enter the elements of matrix B:" << endl;
    for (int i = 0; i < p; i++) {
        for (int j = 0; j < r; j++) {
            cin >> B[i][j];
        }
    }
    // Compute the product matrix C
    int C[n][r] = {0};
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < r; j++) {
```

```
            for (int k = 0; k < m; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    // Output the product matrix C
    cout << "Product of matrices A and B:" << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < r; j++) {
            cout << C[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Example 2.7:

Write a program to find the transpose of a matrix A[n][m].

```
#include <iostream>
using namespace std;
int main() {
    int n, m;
    // Input the dimensions of the matrix
    cout << "Enter the number of rows of matrix A: ";
    cin >> n;
    cout << "Enter the number of columns of matrix A: ";
    cin >> m;
    // Declare matrix A and its transpose
    int A[n][m], transpose[m][n];
    // Input matrix A
    cout << "Enter the elements of matrix A (" << n << "x" << m << "):" << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
```

```
            cin >> A[i][j];
        }
    }
    // Calculate the transpose of matrix A
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            transpose[i][j] = A[j][i];
        }
    }
    // Output the transpose matrix
    cout << "Transpose of matrix A:" << endl;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            cout << transpose[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

10. Practical Applications of Two-Dimensional Arrays

This section illustrates the broad applications of 2D arrays. Explore their versatility across different domains and tasks.

Example 2.8:

Write a program that reads a 3×4 2D array and substitutes every occurrence of 2 with 0.

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

int main() {
    const int rows = 3;
    const int cols = 4;
    int a[rows][cols];
```

```
    // Read the elements of the 2D array
    cout << "Enter the elements of the 3x4 array:" << endl;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cin >> a[i][j];
        }
    }
    // Substitute each occurrence of 2 with 0
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (a[i][j] == 2) {
                a[i][j] = 0;
            }
        }
    }
    // Output the modified array
    cout << "Modified array:" << endl;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cout << a[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Example 2.9:

Write a program to multiply the array elements by 2.

```
#include <iostream>
using namespace std;
int main() {
    const int size = 5;
    int a[size];
```

```
    // Input array elements
    cout << "Enter " << size << " array elements:" << endl;
    for (int i = 0; i < size; i++) {
        cin >> a[i];
    }
    // Multiply each element by 2
    for (int i = 0; i < size; i++) {
        a[i] =a[i]*2;
    }
    // Output the modified array
    cout << "Array after multiplying each element by 2:" << endl;
    for (int i = 0; i < size; i++) {
        cout << a[i] << " ";
    }
    cout << endl;
    return 0;
}
```

Example 2.10:

Write a program to convert a 2D array to 1D array

```
#include <iostream>
using namespace std;
int main() {
    const int rows = 3;
    const int cols = 4;
    int arr2D[rows][cols];
    int arr1D[rows * cols];
    // Input elements into 2D array
    cout << "Enter " << rows * cols << " elements for the 2D array:" << endl;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cin >> arr2D[i][j];
        }
```

```
}
// Convert 2D array to 1D array
int k = 0;
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        arr1D[k++] = arr2D[i][j];
}
 }
```

Example 2.11:

Write a program, to convert 1 D-array that size [16] to 2D-array that size of [4] [4].

```
#include <iostream>
using namespace std;
int main() {
    const int size1D = 16;
    const int rows = 4;
    const int cols = 4;
    int arr1D[size1D];
    int arr2D[rows][cols];
    // Input elements into 1D array
    cout << "Enter " << size1D << " elements for the 1D array:" << endl;
    for (int i = 0; i < size1D; i++)
        cin >> arr1D[i];
    // Convert 1D array to 2D array
    int k = 0;
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            arr2D[i][j] = arr1D[k++];
    // Output the 2D array
    cout << "Converted 2D array:" << endl;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++)
            cout << arr2D[i][j] << " ";
```

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

Exercisers

1. Write a program to calculate the sum of all odd numbers in a 2D array.
2. Write a program to search for a specific value, X, in a 2D array and return its index.
3. Write a program to read an array A[n][m] of numbers and replace each even positive number with 1 and each odd positive number with 0.
4. Write a program to input an array A[n][m] of numbers and find the minimum value in the array.
5. Write a program to swap row1 and row3 in a 4×3 array.

3

2D Square Matrix

1. Introduction

A **square matrix** is a 2D-dimensional array with the same number of rows and columns. The order of a square matrix is given by n; for example, the following square matrix is of order 3.

	[0]	[1]	[2]
[0]	A[0] [0]	A[0] [1]	A[0] [2]
[1]	A[1] [0]	A[1] [1]	A[1] [2]
[2]	A[2] [0]	A[2] [1]	A[2] [2]

It is a special case of a 2D-dimensional array. The same bracket [ ] is used to allocate locations in memory of size $n \times n$. The syntax for declaring a two-dimensional square array is:

```
datatype   arrayName[n][n];
```

The initialization could be performed in different ways:

Method 1:

```
int arr[2][2] = {10, 11 ,12 ,20 }; // the initialization is performed row by row
```

Method 2:

```
int arr[2][2] = {{10, 11} , {21 , 22}};
```

Method 3:

Or you can just allocate a fixed location of an array without assigning values to them. Then the assignment operation can be performed at run time when the user can enter any value.

2. Accessing Square Matrix Elements

2.1. Elements on the Main Diagonal

The elements in the main diagonal have the same index number (i=j).

A[0][0]	A[0][1]	A[0][2]	A[0][3]	A[0][4]
A[1][0]	**A[1][1]**	A[1][2]	A[1][3]	A[1][4]
A[2][0]	A[2][1]	**A[2][2]**	A[2][3]	A[2][4]
A[3][0]	A[3][1]	A[3][2]	**A[3][3]**	A[3][4]
A[4][0]	A[4][1]	A[4][2]	A[4][3]	**A[4][4]**

Example 3.1:

Write a C++ program to read a square array of order 3 by the user, then print the array elements in rows, and find the average of the main diagonal.

```
#include <iostream>
using namespace std;
int main() {
   const int n = 3; // Change this value if you want a different order
   int A[n][n];
   // Input phase
   cout << "Enter the elements of the square array of order " << n << ":\n";
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++) {
         cout << "Enter element at position (" << i << ", " << j << "): ";
         cin >> A[i][j];
      }
   // Output phase - printing the array elements in rows
   cout << "\nArray elements in rows:\n";
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++)
         cout << A[i][j] << " ";
```

```
        cout << "\n";
    }
    // Calculate the average of the main diagonal
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += A[i][i];
    double average = sum / (double) n;
    // Output the average of the main diagonal
    cout << "\nAverage of the main diagonal: " << average << "\n";
    return 0;
}
```

2.2. Elements Above the Main Diagonal

The elements above the main diagonal have the row index less than the column index (i<j).

A[0] [0]	A[0] [1]	A[0] [2]	A[0] [3]	A[0] [4]
A[1] [0]	A[1] [1]	A[1] [2]	A[1] [3]	A[1] [4]
A[2] [0]	A[2] [1]	A[2] [2]	A[2] [3]	A[2] [4]
A[3] [0]	A[3] [1]	A[3] [2]	A[3] [3]	A[3] [4]
A[4] [0]	A[4] [1]	A[4] [2]	A[4] [3]	A[4] [4]

Example 3.2:

Write a C++ program to initialize a square array of order 4 with zeros, then replace the elements above the main diagonal with the value 5.

```
#include <iostream>
using namespace std;
int main() {
    const int n = 4; // Change this value if you want a different order
    int A[n][n];
    // Initialization phase
```

```
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            A[i][j] = 0; // Initialize all elements to 0
    // Replace elements above the main diagonal with 5
    for (int i = 0; i < n; i++)
        for (int j = i + 1; j < n; j++)
            A[i][j] = 5;
    // Output phase - printing the array elements in rows
    cout << "Array elements after replacing above main diagonal with 5:\n";
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            cout << A[i][j] << " ";
        cout << "\n";
    }
    return 0;
}
```

2.3. Elements Below the Main Diagonal

The elements below the main diagonal have the row index greater than the column index (i>j).

A[0] [0]	A[0] [1]	A[0] [2]	A[0] [3]	A[0] [4]
A[1] [0]	A[1] [1]	A[1] [2]	A[1] [3]	A[1] [4]
A[2] [0]	A[2] [1]	A[2] [2]	A[2] [3]	A[2] [4]
A[3] [0]	A[3] [1]	A[3] [2]	A[3] [3]	A[3] [4]
A[4] [0]	A[4] [1]	A[4] [2]	A[4] [3]	A[4] [4]

Example 3.3:

Write a C++ program to read an array of size $n \times n$, where n is equal to 5. Find the largest element among the elements below the main diagonal.

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

```
int main() {
   const int n = 5;
   int A[n][n];
   // Initialization phase
   cout << "Enter the elements of the array of size " << n << "x" << n << ":\n";
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++) {
         cout << "Enter element at position (" << i << ", " << j << "): ";
         cin >> A[i][j];
      }
   // Output the matrix
   cout << "\nThe matrix is:\n";
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++)
         cout << A[i][j] << " ";
      cout << "\n";
   }
   // Finding the largest element below the main diagonal
   int L = A[1][0]; // Initialize with the first element below the main diagonal
   for (int i = 2; i < n; i++)
      for (int j = 0; j < i; j++)
         if (A[i][j] > L)
            L = A[i][j];
   // Output the largest element below the main diagonal
   cout << "\nThe largest element below the main diagonal is: " << L << endl;
   return 0;
}
```

2.4. Elements on the Secondary Diagonal

The elements on the secondary diagonal have the row index plus the column index equal to the matrix size-1, (i+j=n-1).

A[0] [0]	A[0] [1]	A[0] [2]	A[0] [3]	A[0] [4]
A[1] [0]	A[1] [1]	A[1] [2]	A[1] [3]	A[1] [4]
A[2] [0]	A[2] [1]	A[2] [2]	A[2] [3]	A[2] [4]
A[3] [0]	A[3] [1]	A[3] [2]	A[3] [3]	A[3] [4]
A[4] [0]	A[4] [1]	A[4] [2]	A[4] [3]	A[4] [4]

Example 3.4:

Write a C++ program to read a 3×3 array by the user, and find the total summation of the main diagonal elements and the total summation of the secondary diagonal elements.

```
#include <iostream>
using namespace std;
int main() {
   const int n = 3;
   int A[n][n];
   // Input phase
   cout << "Enter the elements of the " << n << "x" << n << " array:\n";
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++) {
         cout << "Enter element at position (" << i << ", " << j << "): ";
         cin >> A[i][j];
      }
   // Output the array
   cout << "\nThe entered array is:\n";
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++)
         cout << A[i][j] << " ";
```

```
        cout << "\n";
    }
    // Calculate the total summation of the main diagonal elements
    int mainDiagonalSum = 0;
    for (int i = 0; i < n; i++)
        mainDiagonalSum += A[i][i];
    // Calculate the total summation of the secondary diagonal elements
    int secondDiagonalSum = 0;
    for (int i = 0; i < n; i++)
        secondDiagonalSum += A[i][n - 1 - i];
    // Output the results
    cout << "\nTotal summation of the main diagonal elements: " << mainDiagonalSum << endl;
    cout << "Total summation of the secondary diagonal elements: " << secondDiagonalSum <<
endl;
    return 0;
}
```

2.5. Elements Above the Secondary Diagonal

The elements above the secondary diagonal have the row index plus the column index is less than to the matrix size -1, (i+j<n-1).

A[0][0]	A[0][1]	A[0][2]	A[0][3]	A[0][4]
A[1][0]	A[1][1]	A[1][2]	A[1][3]	A[1][4]
A[2][0]	A[2][1]	A[2][2]	A[2][3]	A[2][4]
A[3][0]	A[3][1]	A[3][2]	A[3][3]	A[3][4]
A[4][0]	A[4][1]	A[4][2]	A[4][3]	A[4][4]

Example 3.5:

Write a C++ program to set up a **4 × 4** square matrix with all elements initialized to zero. Then, replace the elements above the secondary diagonal with 1.

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

```
const int n = 4;
int main() {
   // Initialize a 4x4 matrix with all elements set to zero
   int A[n][n] = {0};
   // Replace elements above the secondary diagonal with 1
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
         if (i < n - j - 1)
            A[i][j] = 1;
    // Display the resulting matrix
   cout << "Modified Matrix:" << endl;
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++)
         cout << A[i][j] << " ";
      cout << endl;
   }
   return 0;
}
```

2.6. Elements Below the Secondary Diagonal

The elements below the secondary diagonal have the row index plus the column index greater than the matrix size -1, (i+j>n-1).

A[0] [0]	A[0] [1]	A[0] [2]	A[0] [3]	A[0] [4]
A[1] [0]	A[1] [1]	A[1] [2]	A[1] [3]	A[1] [4]
A[2] [0]	A[2] [1]	A[2] [2]	A[2] [3]	A[2] [4]
A[3] [0]	A[3] [1]	A[3] [2]	A[3] [3]	A[3] [4]
A[4] [0]	A[4] [1]	A[4] [2]	A[4] [3]	A[4] [4]

2.7. Corner Elements

The corner elements are in the four corners of a square matrix. The corner elements have a row index equal to 0 or n-1 and a column index equal to 0 or n-1. (i = 0 OR i=n-1) AND (j=0 OR j=n-1).

A[0] [0]	A[0] [1]	A[0] [2]	A[0] [3]	A[0] [4]
A[1] [0]	A[1] [1]	A[1] [2]	A[1] [3]	A[1] [4]
A[2] [0]	A[2] [1]	A[2] [2]	A[2] [3]	A[2] [4]
A[3] [0]	A[3] [1]	A[3] [2]	A[3] [3]	A[3] [4]
A[4] [0]	A[4] [1]	A[4] [2]	A[4] [3]	A[4] [4]

Example 3.6:

Write a C++ program to set up a square matrix of size 4×4. Replace the elements in the corners of the matrix with the sum of all elements in the matrix.

```
#include <iostream>
using namespace std;
const int n = 4;
int main() {
  // Initialize a 4x4 matrix with elements
  int A[n][n] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12},
    {13, 14, 15, 16}
  };
  // Calculate the sum of all elements in the matrix
  int totalSum = 0;
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      totalSum += A[i][j];
```

```
    // Replace the corner elements with the sum
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            if ((i == 0 || i == n - 1) && (j == 0 || j == n - 1))
                A[i][j] = totalSum;
    // Display the resulting matrix
    cout << "Modified Matrix:" << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            cout << A[i][j] << " ";
        cout << endl;
    }
    return 0;
}
```

2.8. Boundary Elements

The boundary elements have a row index equal either to 0 or n-1, or the column index equals either to 0 or n-1, (i=0 OR i=n-1 OR j=0 OR j=n-1).

A[0] [0]	A[0] [1]	A[0] [2]	A[0] [3]	A[0] [4]
A[1] [0]	A[1] [1]	A[1] [2]	A[1] [3]	A[1] [4]
A[2] [0]	A[2] [1]	A[2] [2]	A[2] [3]	A[2] [4]
A[3] [0]	A[3] [1]	A[3] [2]	A[3] [3]	A[3] [4]
A[4] [0]	A[4] [1]	A[4] [2]	A[4] [3]	A[4] [4]

Example 3.7:

Write a C++ program to set up a **4 × 4** square matrix with all elements initialized to zero. Then, replace the elements on the boundary with 1.

```
#include <iostream>
using namespace std;
const int n = 4;
```

```
int main() {
   // Initialize a 4x4 matrix with all elements set to zero
   int A[n][n] = {0};
   // Replace the elements on the boundary with 1
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
         if (i == 0 || i == n - 1 || j == 0 || j == n - 1)
            A[i][j] = 1;
   // Display the resulting matrix
   cout << "Modified Matrix:" << endl;
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++)
         cout << A[i][j] << " ";
      cout << endl;
   }
   return 0;
}
```

Exercisers

1- Write a C++ program that takes a square matrix A of size n as input and determines whether the sum of its boundary elements is greater than the sum of the elements above the secondary diagonal.
2- Write a C++ program that finds the transpose of a square matrix A of size n.
3- Write a C++ program that finds A^2 a square matrix A of size n.
4- Write a C++ program that takes a square matrix A of size n as input and - Replace the above main diagonal elements with the value of '0', and the below main diagonal elements with the value of '1'.
5- Write a C++ program that takes a square matrix A of size n as input and finds the difference between the largest element among the elements above the secondary diagonal and the largest elements among the elements below the secondary diagonal elements.

4

Multidimensional Arrays

1. Introduction

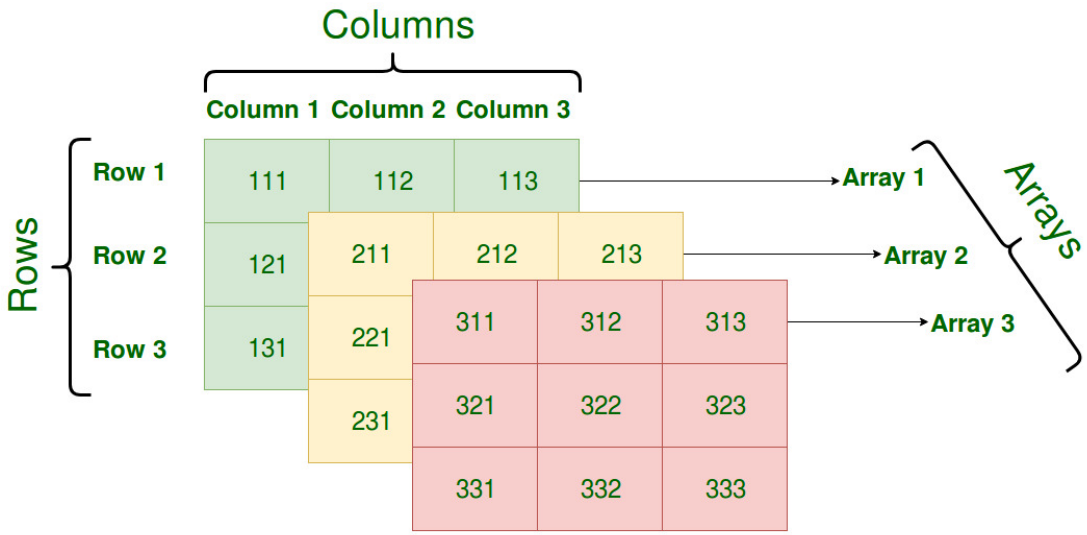
Arrays can have any number of dimensions. In C++ programming, you can create an array of arrays.

To create a multidimensional array in C++, you need to specify the number of dimensions and the size of each dimension.

```
dataType arrayName[size1][size2]...[sizeN];
```

2. 3D Array

The 3D array is a data structure that stores elements in a three-dimensional cuboid-like structure. It can be visualized as a collection of multiple two-dimensional arrays stacked on top of each other. Each element in a 3D array is identified by its three indices: the row index, column index, and depth index.



To declare a 3D array in C++, we need to specify its third dimension along with 2D dimensions. The syntax:

```
dataType arrayName[d][r][c];
```

- dataType: Type of data to be stored in each element.
- arrayName: Name of the array
- d: Number of 2D arrays or Depth of array.
- r: Number of rows in each 2D array.
- c: Number of columns in each 2D array.

The initialization could be performed in different ways:

Method 1:

```
int x[3][5][2] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 30}; // the initialization is performed row by row
```

Method 2:

```
int x[3][5][2] = {
            { {0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9} },
            { {10, 11}, {12, 13}, {14, 15}, {16, 17}, {18, 19} },
            { {20, 21}, {22, 23}, {24, 25}, {26, 27}, {28, 30} },
        };
```

Method 3:

```
int x[3][5][2];
for (int k = 0; k< 3; k++)
  for (int i = 0; i < 5; i++)
    for (int j = 0; j < 2; j++)
      cin>>x[k][i][j];
```

Example 4.1:

Write a C++ program to read an array of [2][3][4], and print the first and second array contents

```
#include <iostream>
using namespace std;
int main() {
```

```
    const int d = 2; // Number of arrays
    const int r = 3; // Number of rows
    const int c = 4; // Number of columns
    int A[d][r][c];
    // Input: Reading the elements of the 3D array
    cout << "Enter the elements of the 3D array:" << endl;
    for (int k= 0; k < d; k++)
        for (int i = 0; i < r; i++)
            for (int j = 0; j < c; j++) {
                cout << "Enter element at [" << k << "][" << i << "][" << j << "]: ";
                cin >> A[k][i][j];
            }
    // Output: Printing the contents of the first array
    cout << "\nContents of the first array:" << endl;
    for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++)
            cout << A[0][i][j] << " ";
        cout << endl;
    }
    // Output: Printing the contents of the second array
    cout << "\nContents of the second array:" << endl;
    for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++)
            cout << A[1][i][j] << " ";
        cout << endl;
    }
    return 0;
}
```

Example 4.2:

Write a C++ program to print the elements of a three-dimensional array named A of size $4 \times 2 \times 3$. Then add the arrays along the depth and print the result on the screen.

```
#include<iostream>
using namespace std;
int main() {
   const int d = 4;
   const int r = 2;
   const int c = 3;
   // Define a three-dimensional array named A
   int A[d][r][c] = {
      {{1, 2, 3}, {4, 5, 6}},
      {{7, 8, 9}, {10, 11, 12}},
      {{13, 14, 15}, {16, 17, 18}},
      {{19, 20, 21}, {22, 23, 24}}
   };
   // Print each matrix within each depth
   for (int k = 0; k < d; k++) {
      cout << "\nMatrix for Depth " << k + 1 << ":" << endl;
      for (int i = 0; i < r; i++) {
         for (int j = 0; j < c; j++) {
            cout << A[k][i][j] << "\t";
         }
         cout << endl;
      }
   }
   // Calculate the arrays along the depth
   int S[r][c] = {0};
   for (int i = 0; i < r; i++) {
         for (int j = 0; j < c; j++) {
           for (int k = 0; k < d; k++) {
            S[i][j] += A[k][i][j];
         }
      }
   }
```

```
   // Output: Printing the sum array
   cout << "\nContents of the sum array:" << endl;
          for (int i = 0; i < r; i++) {
          for (int j = 0; j < c; j++) {
             cout << S[i][j] << "\t";
          }
          cout << endl;
       }
return 0;
}
```

The output of the above code is:

```
Matrix for Depth 1:
1       2       3
4       5       6

Matrix for Depth 2:
7       8       9
10      11      12

Matrix for Depth 3:
13      14      15
16      17      18

Matrix for Depth 4:
19      20      21
22      23      24

Contents of the sum array:
40      44      48
52      56      60
```

Example 4.3:

Write a C++ program to input values for a 3D array, A[2][3][4], and calculate the average of the first rows and the product of the second rows within each depth. Finally, display the computed results on the screen.

```
#include<iostream>
using namespace std;
int main() {
   const int d = 2;
   const int r = 3;
   const int c = 4;
   // Declare a three-dimensional array named A
   int A[d][r][c];
   // Input values for the array A
   cout << "Enter values for the array A[2][3][4]:" << endl;
   for (int k = 0; k < d; k++) {
      cout << "Depth " << k + 1 << ":" << endl;
      for (int i = 0; i < r; i++) {
         for (int j = 0; j < c; j++) {
            cout << "A[" << k << "][" << i << "][" << j << "]: ";
            cin >> A[k][i][j];
         }
      }
   }
   // Print the matrices for each depth
   for (int k = 0; k < d; k++) {
      cout << "\nMatrix for Depth " << k + 1 << ":" << endl;
      for (int i = 0; i < r; i++) {
         for (int j = 0; j < c; j++)
            cout << A[k][i][j] << "\t";
         cout << endl;
      }
```

```
    }
    // Calculate and display the average of the first rows and the product of the second rows
within each depth
    float sumFirstRow[d] = {0};
    float averageFirstRow[d] = {0};
    int productSecondRow[d];
    for (int k = 0; k < d; k++) {
        // Calculate the average of the first row
        for (int j = 0; j < c; j++) {
            sumFirstRow[k] += A[k][0][j];
        }
        averageFirstRow[k] = sumFirstRow[k] / (float) c;
        // Calculate the product of the second row
        productSecondRow[k]=1;
        for (int j = 0; j < c; j++) {
            productSecondRow[k] *= A[k][1][j];
        }
        // Display the results for each depth
        cout << "\nResults for Depth " << k + 1 << ":" << endl;
        cout << "Average of the first row: " << averageFirstRow[k] << endl;
        cout << "Product of the second row: " << productSecondRow[k] << endl;
    }
    return 0;
}
```

The output for the above code is:

```
Enter values for the array A[2][3][4]:
Depth 1:
A[0][0][0]: 1
A[0][0][1]: 2
A[0][0][2]: 3
A[0][0][3]: 4
```

A[0][1][0]: 5
A[0][1][1]: 6
A[0][1][2]: 2
A[0][1][3]: 1
A[0][2][0]: 2
A[0][2][1]: 3
A[0][2][2]: 6
A[0][2][3]: 5
Depth 2:
A[1][0][0]: 4
A[1][0][1]: 2
A[1][0][2]: 2
A[1][0][3]: 1
A[1][1][0]: 3
A[1][1][1]: 6
A[1][1][2]: 7
A[1][1][3]: 1
A[1][2][0]: 2
A[1][2][1]: 5
A[1][2][2]: 1
A[1][2][3]: 2

Matrix for Depth 1:

1	2	3	4
5	6	2	1
2	3	6	5

Matrix for Depth 2:

4	2	2	1
3	6	7	1
2	5	1	2

Results for Depth 1:

Average of the first row: 2.5

Product of the second row: 60

Results for Depth 2:

Average of the first row: 2.25

Product of the second row: 126

Example 4.4:

Write a C++ program to declare a 3D array containing **4** arrays, each with a size of 2×3. Calculate the average of each column within each depth and display the results on the screen.

```
#include<iostream>
using namespace std;
int main() {
   const int d = 4;
   const int r = 2;
   const int c = 3;
   // Define a three-dimensional array named A
   int A[d][r][c] = {
      {{1, 2, 3}, {4, 5, 6}},
      {{7, 8, 9}, {10, 11, 12}},
      {{13, 14, 15}, {16, 17, 18}},
      {{19, 20, 21}, {22, 23, 24}}
   };
   // Print each matrix within each depth
   for (int k = 0; k < d; k++) {
      cout << "\nMatrix for Depth " << k + 1 << ":" << endl;
      for (int i = 0; i < r; i++) {
         for (int j = 0; j < c; j++) {
            cout << A[k][i][j] << "\t";
         }
```

```
            cout << endl;
        }
    }
    // Calculate the sum of each column in each depth
    int columnSum[d][c] = {0};
    for (int k = 0; k < d; k++) {
        for (int i = 0; i < r; i++) {
            for (int j = 0; j < c; j++) {
                columnSum[k][j] += A[k][i][j];
            }
        }
    }
    // Calculate the average of each column in each depth
    float average[d][c] = {0};
    for (int k = 0; k < d; k++) {
        for (int j = 0; j < c; j++) {
            average[k][j] = static_cast<float>(columnSum[k][j]) / r;
        }
    }
    // Print the result on the screen
    cout << "\nSum and Average of each column in each depth:" << endl;
    for (int k = 0; k < d; k++) {
        cout << "\nDepth " << k + 1 << ":" << endl;
        for (int j = 0; j < c; j++) {
            cout << "Column " << j + 1 << ": Sum = " << columnSum[k][j] << ", Average = " <<
average[k][j] << endl;
        }
    }
    return 0;
}
```

The output for the above code is:

```
Matrix for Depth 1:
1       2       3
4       5       6

Matrix for Depth 2:
7       8       9
10      11      12

Matrix for Depth 3:
13      14      15
16      17      18

Matrix for Depth 4:
19      20      21
22      23      24

Sum and Average of each column in each depth:

Depth 1:
Column 1: Sum = 5, Average = 2.5
Column 2: Sum = 7, Average = 3.5
Column 3: Sum = 9, Average = 4.5

Depth 2:
Column 1: Sum = 17, Average = 8.5
Column 2: Sum = 19, Average = 9.5
Column 3: Sum = 21, Average = 10.5

Depth 3:
Column 1: Sum = 29, Average = 14.5
```

Column 2: Sum = 31, Average = 15.5

Column 3: Sum = 33, Average = 16.5

Depth 4:

Column 1: Sum = 41, Average = 20.5

Column 2: Sum = 43, Average = 21.5

Column 3: Sum = 45, Average = 22.5

5

Strings

1. Introduction

In C++ strings of characters are implemented as an array of characters. In addition, a special null character, represented by \0, is appended to the end of the string to indicate the end of the string.

```
char String-name [size];
```

for example.

```
char str [5] = "ABCD";
```

gives

str [0] : 'A'
str [1] : 'B'
str [2] : 'C'
str [3] : 'D'
str [4] : '\0' ↔ *null*

Example 5.1:

Write a C++ program to read a string, then print it character by character:

```
#include <iostream>
using namespace std;
int main() {
   char s[] = "ABCD";
   cout << "Your String is: " << s << endl;
   for (int i = 0; i < 5; i++) {
      cout << "s[" << i << "] is: " << s[i] << endl;
   }
   return 0;
}
```

The output of the above program is:

```
Your String is: ABCD
s[0] is: A
s[1] is: B
s[2] is: C
s[3] is: D
s[4] is:
```

Example 5.2:

Write a C++ program to convert each lowercase letter to an uppercase letter.

```
#include <iostream>
#include <cstring>
#include <cctype>
using namespace std;
int main() {
   char s[] = "abcd";
   cout << s << endl;
   for (int i = 0; i < strlen(s); i++) {
      s[i] = toupper(s[i]);
   }
   cout << s;
   return 0;
}
```

For the above program:

- <iostream> is used for input/output.
- <cstring> is used for string manipulation functions like **strlen**.
- <cctype> is used for character classification and conversion functions like **toupper**.
- The loop runs from 0 to the length of the string s (using **strlen(s)**).
- **toupper** function is used to convert each character to uppercase.

The output of the above code is:

```
abcd
ABCD
```

2. Member Function of String

The string library has many member functions of string like:

Member Function	Functionality	Example
strlen (string)	Return the length of the string.	a [ ] = "abcd"; cout << strlen (a);
strcpy (string2, string1)	Copy the content of the 1st string into the :2nd string.	char a[ ]= "abcd", b[ ]=" "; strcpy (b,a); cout << a<< b; The output is: abcdabcd
strcat (string 1, string2)	Append the content of the 2nd string into the end of the 1st string	char a[ ]="abcd",b[]="1234"; strcat (a , b) ; cout << a<< b; The output is: abcd1234 1234
strcmp (stringl, string2)	Return 0 if the 1st string is equal to the 2nd string. Return a positive number if the 1st string is greater than the 2nd string. Return a negative number if the 1st string is smaller than the 2nd strin .	char a[ ]="abcd", b[ ]="abcd"; cout << strcmp (a, b); The output is: 0 if a== b + if a> b - if a< b

3. stdlib Library

The stdlib library has many member functions of string like:

Member Function	Functionality	Example
atoi (a)	Converts string to int type	int i; char a [ ] = "1234"; i = atoi (a);

atof (a)	Converts string to float type	float f; char a [ ] = "12.34"; f = atof (a);
itoa (i , a , 10);	Converts integer number to alphabet (char or string type)	int i = 1234; char a [ ] = ""; alphabet (char or string type). cout << itoa (i, a, 10);

For the above functions #include <cstdlib> must be included in the header.

Example 5.3:

Write a C++ program to check each character in the string to convert it to a lowercase letter if it's an uppercase letter and convert it to an uppercase letter if it's a lower one.

```
#include <iostream>
#include <cstring>
#include <cctype>
using namespace std;
int main() {
   string input;
   // Read the string
   cout << "Enter a string: ";
   cin >> input;
   // Iterate through each character
   for (int i = 0; i < input.length(); i++) {
      // Check if the character is lowercase
      if (islower(input[i])) {
         // Convert to uppercase
         input[i] = toupper(input[i]);
      }
      // Check if the character is uppercase
      else if (isupper(input[i])) {
         // Convert to lowercase
```

```
            input[i] = tolower(input[i]);
        }
        // Otherwise, leave it unchanged
    }
    // Output the modified string
    cout << "Modified string: " << input << endl;
    return 0;
}
```

The output of the above code is:

```
Enter a string: IrAq
Modified string: iRaQ
```

6

Structures

1. Introduction

Suppose that you want to write a program to process student data. A student record consists of, among other things, the student's name, student ID, GPA, courses taken, and course grades. Thus, various components are associated with a student. However, these components are all of different types. For example, the student's name is a string, and the GPA is a floating-point number. Because these components are of different types, you cannot use an array to group all of the items associated with a student. C++ provides a structured data type called struct to group items of different types. Grouping components that are related but of different types offers several advantages. For example, a single variable can pass all the components as parameters to a function.

A struct is a collection of a fixed number of components in which the components are accessed by name. The components may be of different types.

The components of a struct are called the members of the struct. The general syntax of a struct in C++ is:

```
struct structName
{
        dataType1 identifier1;
        dataType2 identifier2;
        .
        .
        .
       dataTypen identifiern;
};
```

The statement:

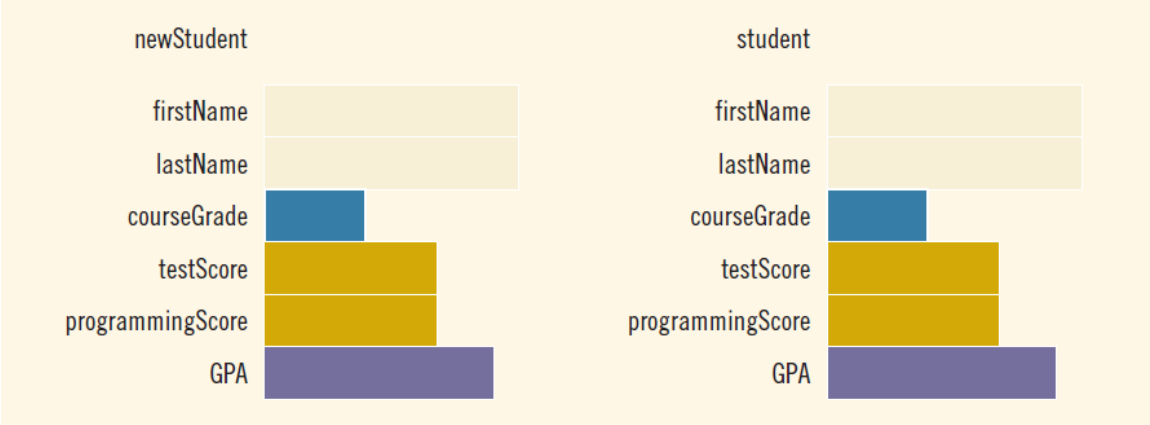
```
struct employeeType
{
        string firstName;
        string lastName;
         string address;
        double salary;
        string deptID;
};
```

defines a **struct employeeType** with five members. The members **firstName**, **lastName**, **address**, and **deptID** are of type **string**, and the member **salary** is of type **double**.

Once a data type is defined, you can declare variables of that type. Let us first define a **struct** type, **studentType**, and then declare variables of that type.

```
        struct studentType
        {
        string firstName;
        string lastName;
        char courseGrade;
        int testScore;
        int programmingScore;
        double GPA;
        };
        //variable declaration
        studentType newStudent;
        studentType student;
```

These statements declare two struct variables, **newStudent** and **student**, of type **studentType**. The memory allocated is large enough to store **firstName**, **lastName**, **courseGrade**, **testScore**, **programmingScore**, and **GPA**



Note that, you can also declare struct variables when you define the **struct**. For example, consider the following statements:

```
struct studentType
        {
        string firstName;
        string lastName;
        char courseGrade;
        int testScore;
        int programmingScore;
        double GPA;
        } tempStudent;
```

These statements define the struct **studentType** and also declare **tempStudent** to be a variable of type **studentType**.

2. Accessing struct Members

To access a structure member (component), you use the struct variable name together with the member name; these names are separated by a dot (period). The syntax for accessing a struct member is:

```
structVariableName.memberName
```

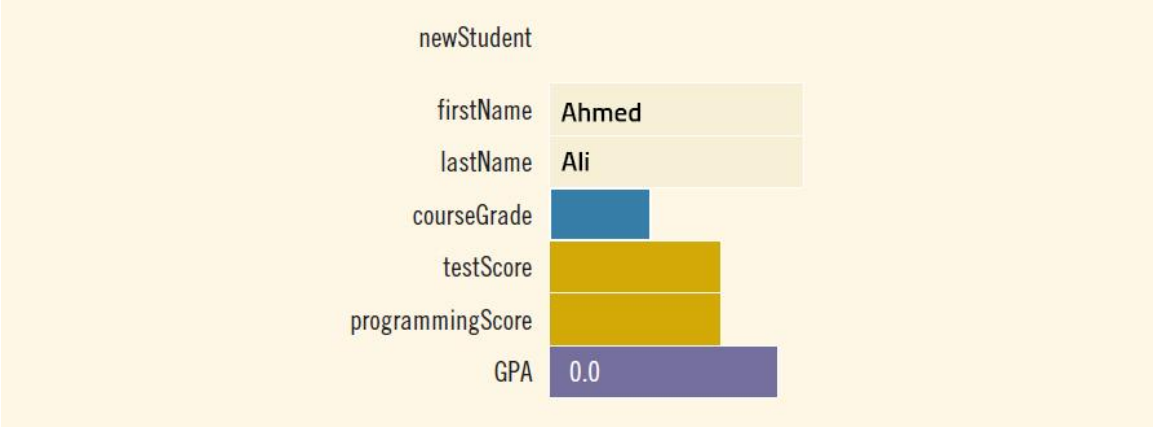
Suppose you want to initialize the member **GPA** of **newStudent** to **0.0**. The following statement accomplishes this task:

```
newStudent.GPA = 0.0;
```

Similarly, the statements:

```
newStudent.firstName = "Ahmed";
newStudent.lastName = "Ali";
```

store "Ahmed" in the member **firstName** and "Ali" in the member **lastName** of **newStudent**.



Example 6.1:

Write a C++ program to define a structure named 'student' which has 4 items (student's name, student's phone number, student's blood type, and student's average). Let the user read the values for the struct members and print them.

```
#include <iostream>
using namespace std;
// Define the structure
struct student {
   string name;
   string phone_number;
   string blood_type;
   float average;
};
int main() {
   // Declare a variable of type 'student'
   student s;
   // Prompt the user to enter values for the struct members
   cout << "Enter student's name: ";
   cin >> s.name;
   cout << "Enter student's phone number: ";
   cin >> s.phone_number;
```

```
    cout << "Enter student's blood type: ";
    cin >> s.blood_type;
    cout << "Enter student's average: ";
    cin >> s.average;
    // Print the values of the struct members
    cout << "\nStudent's Information:" << endl;
    cout << "Name: " << s.name << endl;
    cout << "Phone Number: " << s.phone_number << endl;
    cout << "Blood Type: " << s.blood_type << endl;
    cout << "Average: " << s.average << endl;
    return 0;
}
```

The output of the above code is:

```
Enter student's name: Ahmed
Enter student's phone number: 7901999999
Enter student's blood type: A+
Enter student's average: 74.25

Student's Information:
Name: Ahmed
Phone Number: 7901999999
Blood Type: A+
Average: 74.25
```

Example 6.2:

Write a C++ program to define a structure which has 3 members (student's name, student's height, and student's ID). Suppose you have two students record and you need to choose among the tallest one for a school's basketball team.

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

```
// Define the structure for student record
struct Student {
    string name;
    double height;
    int ID;
};
int main() {
    // Declare variables for two student records
    Student student1, student2;
    // Input details for student 1
    cout << "Enter details for student 1:\n";
    cout << "Name: ";
    cin >> student1.name;
    cout << "Height (in meters): ";
    cin >> student1.height;
    cout << "ID: ";
    cin >> student1.ID;
    // Input details for student 2
    cout << "\nEnter details for student 2:\n";
    cout << "Name: ";
    cin >> student2.name;
    cout << "Height (in meters): ";
    cin >> student2.height;
    cout << "ID: ";
    cin >> student2.ID;
    // Choose the tallest student for the basketball team
    Student tallest_student;
    if (student1.height > student2.height) {
        tallest_student = student1;
    } else {
        tallest_student = student2;
    }
```

```
    // Output the details of the chosen student
    cout << "\nThe tallest student for the basketball team is:\n";
    cout << "Name: " << tallest_student.name << endl;
    cout << "Height: " << tallest_student.height << " meters" << endl;
    cout << "ID: " << tallest_student.ID << endl;
    return 0;
}
```

The output of the above code is:

```
Enter details for student 1:
Name: Ahmed
Height (in meters): 1.78
ID: 2

Enter details for student 2:
Name: Rami
Height (in meters): 1.9
ID: 1

The tallest student for the basketball team is:
Name: Rami
Height: 1.9 meters
ID: 1
```

Example 6.3:

Create a C++ program that defines a structure containing three members: the player's name, height, and weight. Determine the player's Body Mass Index (BMI) and display their corresponding weight category.

Note that, BMI is a measure used to estimate whether a person has a healthy body weight for their height. BMI is calculated by dividing a person's weight in kilograms by the square of their height in meters. The formula is:

$$BMI = \frac{weigh}{height^2}$$

Here's what the BMI categories typically mean for adults:

- BMI below 18.5: Underweight
- BMI 18.5—24.9: Normal weight
- BMI 25—29.9: Overweight
- BMI 30 or higher: Obesity

```
#include <iostream>
#include <cmath>
using namespace std;
// Structure definition for player
struct Player {
   string name;
   double height; // in meters
   double weight; // in kilograms
};
int main() {
   // Create a player instance
   Player player;
   // Input player's data
   cout << "Enter player's name: ";
   cin >> player.name;
   cout << "Enter player's height (in meters): ";
   cin >> player.height;
   cout << "Enter player's weight (in kilograms): ";
   cin >> player.weight;
   // Calculate BMI
   double bmi = player.weight / pow(player.height, 2);
   // Determine weight category
   string category;
   if (bmi < 18.5) {
      category = "Underweight";
   } else if (bmi >= 18.5 && bmi <= 24.9) {
```

```
        category = "Normal weight";
    } else if (bmi >= 25 && bmi <= 29.9) {
        category = "Overweight";
    } else {
        category = "Obesity";
    }
    // Display results
    cout << "\nPlayer: " << player.name << endl;
    cout << "Height: " << player.height << " meters" << endl;
    cout << "Weight: " << player.weight << " kilograms" << endl;
    cout << "BMI: " << bmi << endl;
    cout << "Weight category: " << category << endl;
    return 0;
}
```

The output of the above code is:

```
Enter player's name: Ahmed
Enter player's height (in meters): 1.78
Enter player's weight (in kilograms): 75

Player: Ahmed
Height: 1.78 meters
Weight: 75 kilograms
BMI: 23.6713
Weight category: Normal weight
```

Example 6.4:

Write a C++ program that asks the user to input the coordinates of a point in a 2D coordinate system. The program should then determine and display the quadrant in which the point lies or indicate if it lies on an axis or at the origin.

```
#include <iostream>
using namespace std;
// Define a struct to represent a point
struct Coordinate {
   double x;
   double y;
};
int main() {
   // Declare a variable of type Coordinate to store the coordinates
   Coordinate point;
   // Ask user to input coordinates
   cout << "Enter the x-coordinate: ";
   cin >> point.x;
   cout << "Enter the y-coordinate: ";
   cin >> point.y;
   // Determine quadrant or axis
   if (point.x > 0 && point.y > 0)
      cout << "The point lies in the first quadrant." << endl;
   else if (point.x < 0 && point.y > 0)
      cout << "The point lies in the second quadrant." << endl;
   else if (point.x < 0 && point.y < 0)
      cout << "The point lies in the third quadrant." << endl;
   else if (point.x > 0 && point.y < 0)
      cout << "The point lies in the fourth quadrant." << endl;
   else if (point.x == 0 && point.y != 0)
      cout << "The point lies on the y-axis." << endl;
   else if (point.x != 0 && point.y == 0)
      cout << "The point lies on the x-axis." << endl;
   else
      cout << "The point lies at the origin." << endl;
   return 0;
}
```

The output of the above code is:

```
Enter the x-coordinate: -2.1
Enter the y-coordinate: -5
The point lies in the third quadrant.
```

Or:

```
Enter the x-coordinate: 1
Enter the y-coordinate: 0
The point lies on the x-axis.
```

Or:

```
Enter the x-coordinate: 0
Enter the y-coordinate: 5
The point lies on the y-axis.
```

Or:

```
Enter the x-coordinate: 0
Enter the y-coordinate: 0
The point lies at the origin.
```

Or:

```
Enter the x-coordinate: -1.5
Enter the y-coordinate: 2
The point lies in the second quadrant.
```

7

Arrays of Structures

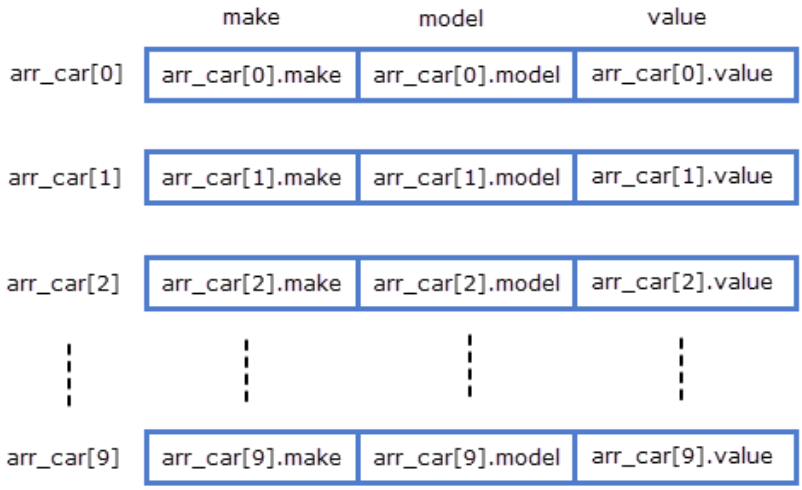
1. Introduction

An array of structures in C++ can be defined as the collection of multiple structures variables where each variable contains information about different entities. In other words, the array of structures in C++ are used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures. For example,

```
struct car
{
   char make[20];
   char model[30];
   int value;
};
```

Here is how we can declare an array of structure car.

```
car arr_car[10];
```



Here arr_car is an array of 10 elements where each element is of type car. We can use arr_car to store 10 structure variables of type car. To access individual elements we will use subscript notation ([]) and to access the members of each element we will use dot (.)

Example 7.1:

What is a C++ program that creates an array of structures to store information about students' names, roll numbers, and marks, and then prints out the details entered by the user.

```
#include <iostream>
using namespace std;
// Define a structure for student
struct Student {
   char name[50]; // Assuming maximum length of name is 50 characters
   int rollNumber;
   float marks;
};
int main() {
   int n;
   cout << "Enter the number of students: ";
   cin >> n;
   // Declare an array of structures
   Student students[n];
   // Input details for each student
   for (int i = 0; i < n; i++) {
      cout << "Enter details for student " << i + 1 << ":" << endl;
      cout << "Name: ";
      cin >> students[i].name;
      cout << "Roll Number: ";
      cin >> students[i].rollNumber;
      cout << "Marks: ";
      cin >> students[i].marks;
```

```
    }
    // Print details of all students
    cout << "Details of students entered:" << endl;
    for (int i = 0; i < n; i++) {
        cout << "Student " << i + 1 << ":" << endl;
        cout << "Name: " << students[i].name << endl;
        cout << "Roll Number: " << students[i].rollNumber << endl;
        cout << "Marks: " << students[i].marks << endl;
    }
    return 0;
}
```

The output of the above code is:

```
Enter the number of students: 2
Enter details for student 1:
Name: Ahmed
Roll Number: 14
Marks: 85
Enter details for student 2:
Name: Ali
Roll Number: 45
Marks: 90
Details of students entered:
Student 1:
Name: Ahmed
Roll Number: 14
Marks: 85
Student 2:
Name: Ali
Roll Number: 45
Marks: 90
```

Example 7.2:

Write a C++ program that asks the user to input details for four students, including their names, IDs, stages, and marks for five subjects each. The program should then calculate and display the average marks for each student along with their details.

```
#include <iostream>
using namespace std;
// Structure to store student details
struct Student {
   char name[50];
   int id;
   int stage;
   float marks[5];
};
int main() {
   const int NUM_STUDENTS = 4;
   const int NUM_SUBJECTS = 5;
   // Array to store student details
   Student A[NUM_STUDENTS];
   // Input details for each student
   for (int i = 0; i < NUM_STUDENTS; i++) {
      cout << "Enter details for Student " << i + 1 << ":\n";
      cout << "Name: ";
      cin >> A[i].name;
      cout << "ID: ";
      cin >> A[i].id;
      cout << "Stage: ";
      cin >> A[i].stage;
      cout << "Enter marks for 5 subjects:\n";
      for (int j = 0; j < NUM_SUBJECTS; j++) {
         cout << "Subject " << j + 1 << ": ";
         cin >> A[i].marks[j];
      }
```

```
    }
     // Calculate and display average marks for each student
    cout << "\n\nStudent Details and Average Marks:\n";
    for (int i = 0; i < NUM_STUDENTS; i++) {
        float totalMarks = 0;
        for (int j = 0; j < NUM_SUBJECTS; j++) {
            totalMarks += A[i].marks[j];
        }
        float averageMarks = totalMarks / NUM_SUBJECTS;
        cout << "Name: " << A[i].name << endl;
        cout << "ID: " << A[i].id << endl;
        cout << "Stage: " << A[i].stage << endl;
        cout << "Average Marks: " << averageMarks << endl << endl;
    }
        return 0;
}
```

Example 7.3:

Consider a class consisting of 10 students, each identified by their name and their marks in three subjects: 'Linear Algebra', 'Ordinary Differential Equations (ODE)', and 'Graph Theory'. Write a C++ program to achieve the following tasks:

1. Determine the number of students who pass in each subject.
2. Identify the highest achiever among the ten students in each subject.
3. Display the name of each student along with their average marks.

```
#include <iostream>
#include <string>

using namespace std;
```

```
// Define a struct for student
struct Student {
    string name;
    double marks_linear_algebra;
    double marks_ode;
    double marks_graph_theory;
    double avg_marks;
};

int main() {
    int n;
    cout << "Enter the number of students: ";
    cin >> n;

    // Dynamic allocation of students array
    Student students[n];

    // Input marks and names for each student
    cout << "Enter the details for each student:" << endl;
    for (int i = 0; i < n; i++) {
        cout << "Student " << i + 1 << ":" << endl;
        cout << "Name: ";
        cin >> students[i].name;
        cout << "Marks in Linear Algebra: ";
        cin >> students[i].marks_linear_algebra;
        cout << "Marks in ODE: ";
        cin >> students[i].marks_ode;
        cout << "Marks in Graph Theory: ";
        cin >> students[i].marks_graph_theory;

        // Calculate average marks for each student
```

```
    students[i].avg_marks = (students[i].marks_linear_algebra + students[i].marks_ode +
students[i].marks_graph_theory) / 3.0;

    cout << endl;
  }

  // Task 1: Determine the number of students who pass in each subject
  int pass_linear_algebra = 0, pass_ode = 0, pass_graph_theory = 0;
  for (int i = 0; i < n; i++) {
    if (students[i].marks_linear_algebra >= 50) pass_linear_algebra++;
    if (students[i].marks_ode >= 50) pass_ode++;
    if (students[i].marks_graph_theory >= 50) pass_graph_theory++;
  }

  // Task 2: Identify the highest achiever among the students in each subject
  Student highest_linear_algebra = students[0];
  Student highest_ode = students[0];
  Student highest_graph_theory = students[0];

  for (int i = 1; i < n; i++) {
    if (students[i].marks_linear_algebra > highest_linear_algebra.marks_linear_algebra)
      highest_linear_algebra = students[i];
    if (students[i].marks_ode > highest_ode.marks_ode)
      highest_ode = students[i];
    if (students[i].marks_graph_theory > highest_graph_theory.marks_graph_theory)
      highest_graph_theory = students[i];
  }

  // Task 3: Display the name of each student along with their average marks
  cout << "Student Name\t\tAverage Marks" << endl;
  for (int i = 0; i < n; i++) {
    cout << students[i].name << "\t\t" << students[i].avg_marks << endl;
```

```
    }

    // Display results of Task 1 and Task 2
    cout << endl << "Number of students passing in each subject:" << endl;
    cout << "Linear Algebra: " << pass_linear_algebra << endl;
    cout << "ODE: " << pass_ode << endl;
    cout << "Graph Theory: " << pass_graph_theory << endl;
    cout << endl << "Highest achievers in each subject:" << endl;
    cout << "Linear Algebra: " << highest_linear_algebra.name << endl;
    cout << "ODE: " << highest_ode.name << endl;
    cout << "Graph Theory: " << highest_graph_theory.name << endl;

    return 0;
}
```

The output of the above code is:

```
Enter the number of students: 2
Enter the details for each student:
Student 1:
Name: Ali
Marks in Linear Algebra: 90
Marks in ODE: 45
Marks in Graph Theory: 75

Student 2:
Name: Noor
Marks in Linear Algebra: 95
```

Marks in ODE: 60

Marks in Graph Theory: 45

Student Name	Average Marks
Ali	70
Noor	66.6667

Number of students passing in each subject:

Linear Algebra: 2

ODE: 1

Graph Theory: 1

Highest achievers in each subject:

Linear Algebra: Noor

ODE: Noor

Graph Theory: Ali

8

Functions

1. Introduction

A function is a block of code that performs a specific task.
Dividing a complex problem into smaller tasks makes our program easy to understand and reusable.
There are two types of functions:

1. Standard Library Functions: Predefined in C++.
2. User-defined Function: Created by users.

2. Predefined Functions

In C++, predefined functions play a fundamental role in simplifying programming tasks by providing a set of commonly used functionalities. These functions are part of the standard library and cover a wide range of operations, from basic arithmetic to complex algorithms. The following table gives some examples of these functions:

Function	Header File	Purpose
abs(x)	<cmath>	Returns the absolute value of its argument: abs(-7) = 7
ceil(x)	<cmath>	Returns the smallest whole number that is not less than x: ceil(56.34) = 57.0
cos(x)	<cmath>	Returns the cosine of angle: x: cos(0.0) = 1.0
exp(x)	<cmath>	Returns e^x, where e = 2.718: exp(1.0) = 2.71828
fabs(x)	<cmath>	Returns the absolute value of its argument: fabs(-5.67) = 5.67

floor(x)	<cmath>	Returns the largest whole number that is not greater than x:floor(45.67) = 45.00
islower(x)	<cctype>	Returns true if x is a lowercase letter; otherwise, it returns false; islower('h') is true
isupper(x)	<cctype>	Returns true if x is a uppercase letter; otherwise, it returns false; isupper('K') is true
pow(x, y)	<cmath>	Returns x^y; if x is negative, y must be a whole number: pow(0.16, 0.5) = 0.4
sqrt(x)	<cmath>	Returns the nonnegative square root of x; x must be nonnegative: sqrt(4.0) = 2.0
tolower(x)	<cctype>	Returns the lowercase value of x if x is uppercase; otherwise, it returns x
toupper(x)	<cctype>	Returns the uppercase value of x if x is lowercase; otherwise, it returns x

Example 8.1:

This example shows a C++ program to find the square root of a number

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
   double number, squareRoot;
   number = 25.0;
   // sqrt() is a library function to calculate the square root
   squareRoot = sqrt(number);
   cout << "Square root of " << number << " = " << squareRoot;
   return 0;
}
```

Output

```
Square root of 25 = 5
```

3. C++ User-Defined Functions

C++ allows the programmer to define their own function.

A user-defined function groups code to perform a specific task and that group of code is given a name (identifier).

When the function is called from any part of the program, it all executes the codes defined in the body of the function.

3.1 C++ Function Declaration

The syntax to declare a function is:

```
returnType   functionName (parameter1, parameter2,...)
{
   // function body
}
```

Here's an example of a function declaration.

```
// function declaration
void greet() {
  cout << "Hello World";
}
```

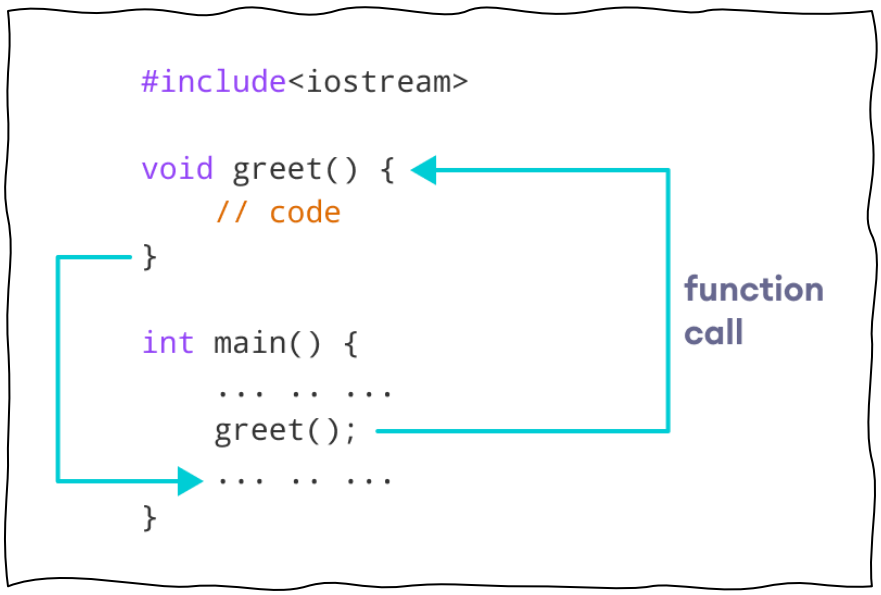
Here,

- the name of the function is **greet()**
- the return type of the function is **void**
- the empty parentheses mean it doesn't have any parameters
- the function body is written inside **{}**

3.2 Calling a Function

In the above program, we have declared a function named **greet()**. To use the **greet()** function, we need to call it. Here's how we can call the above **greet()** function.

```
int main() {
    // calling a function
  greet();
}
```



How Function works in C++

Example 8.2:

In this C++ code, a function named "greet()" is declared and called to print "Hello there!" to the console.

```
#include <iostream>
using namespace std;
// declaring a function
void greet() {
  cout << "Hello there!";
}
int main() {
  // calling the function
  greet();
  return 0;
}
```

Output

```
Hello there!
```

3.3 Function Parameters

As mentioned above, a function can be declared with parameters (arguments). A parameter is a value that is passed when declaring a function.
For example, let us consider the function below:

```
void printNum(int num) {
   cout << num;
}
```

Here, the int variable num is the function parameter.

We pass a value to the function parameter while calling the function.

```
int main() {
   int n = 7;
   // calling the function
   // n is passed to the function as argument
   printNum(n);
   return 0;
}
```

Example 8.3:

This example shows how to use a function with parameters

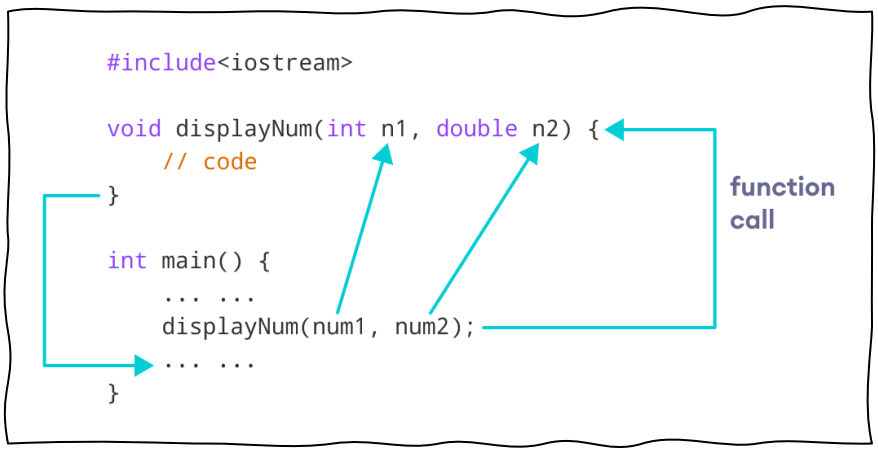
```
// program to print a text
#include <iostream>
using namespace std;
// display a number
void displayNum(int n1, float n2) {
   cout << "The int number is " << n1;
   cout << "The double number is " << n2;
}
int main() {
    int num1 = 5;
    double num2 = 5.5;
   // calling the function
   displayNum(num1, num2);
   return 0;
}
```

Output

```
The int number is 5
The double number is 5.5
```

In the above program, we have used a function that has one **int** parameter and one **double** parameter.

We then pass **num1** and **num2** as arguments. These values are stored by the function parameters **n1** and **n2** respectively.



C++ function with parameters

Note: The type of arguments passed while calling the function must match with the corresponding parameters defined in the function declaration.

3.4 Return Statement

In the above programs, we have used void in the function declaration. For example,

```
void displayNumber() {
  // code
}
```

This means the function is not returning any value.

It's also possible to return a value from a function. For this, we need to specify the **returnType** of the function during function declaration.

Then, the **return** statement can be used to return a value from a function. For example,

```
int add (int a, int b) {
  return (a + b);
}
```

- Here, we have the data type **int** instead of **void**. This means that the function returns an **int** value.
- The code **return (a + b)**; returns the sum of the two parameters as the function value.
- The **return** statement denotes that the function has ended. Any code after **return** inside the function is not executed.

Example 8.4:

This example defines a function to add two numbers

```
// program to add two numbers using a function
#include <iostream>
using namespace std;
// declaring a function
int add(int a, int b) {
   return (a + b);
}
int main() {
   int sum;
   // calling the function and storing
   // the returned value in sum
   sum = add(100, 78);
   cout << "100 + 78 = " << sum << endl;
   return 0;
}
```

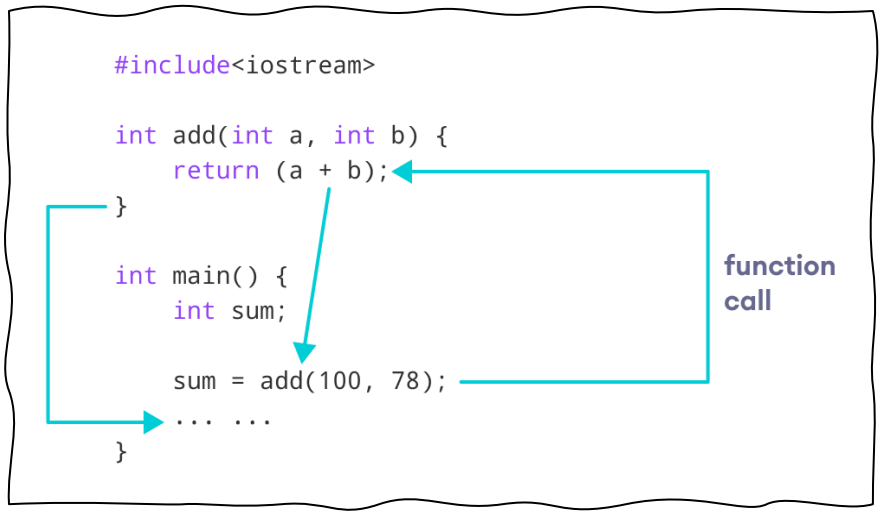
Output

```
100 + 78 = 178
```

In the above program, the **add()** function is used to find the sum of two numbers.

We pass two **int** literals **100** and **78** while calling the function.

We store the returned value of the function in the variable **sum**, and then we print it.



Working of C++ Function with return statement

Notice that sum is a variable of int type. This is because the return value of add() is of int type.

3.5 Function Prototype

In C++, the code of function declaration should be before the function call. However, if we want to define a function after the function call, we need to use the function prototype. For example,

```
// function prototype
void add(int, int);
int main() {
  // calling the function before declaration.
  add(5, 3);
  return 0;
}
// function definition
void add(int a, int b) {
  cout << (a + b);
}
```

In the above code, the function prototype is:

```
void add(int, int);
```

This provides the compiler with information about the function name and its parameters. That's why we can use the code to call a function before the function has been defined.
The syntax of a function prototype is:

```
returnType functionName(dataType1, dataType2, ...);
```

Example 8.5:

This example shows the C++ function prototype

```
// using function definition after main() function
// function prototype is declared before main()
#include <iostream>
using namespace std;
// function prototype
int add(int, int);
int main() {
   int sum;
   // calling the function and storing
   // the returned value in sum
   sum = add(100, 78);
   cout << "100 + 78 = " << sum << endl;
   return 0;
}
// function definition
int add(int a, int b) {
   return (a + b);
}
```

Output

```
100 + 78 = 178
```

The above program is nearly identical to **Example 8.4**. The only difference is that here, the function is defined **after** the function call.
That's why we have used a function prototype in this example.

Benefits of Using User-Defined Functions

- Functions make the code reusable. We can declare them once and use them multiple times.
- Functions make the program easier as each small task is divided into a function.
- Functions increase readability.

Example 8.6:

Write a C++ function to find the average of 3 numbers, let the user read the 3 numbers, and print the result inside the function.

```
#include <iostream>
using namespace std;
// Function to calculate the average of three numbers
double findAverage(double num1, double num2, double num3) {
   // Calculate the average
   return (num1 + num2 + num3) / 3.0;
}
int main() {
   // Variables to store the three numbers
   double num1, num2, num3;
   // Ask the user to input the three numbers
   cout << "Enter the first number: ";
   cin >> num1;
   cout << "Enter the second number: ";
   cin >> num2;
   cout << "Enter the third number: ";
   cin >> num3;
   // Calculate the average
   double average = findAverage(num1, num2, num3);
   // Print the result
```

```
    cout << "The average of " << num1 << ", " << num2 << ", and " << num3 << " is: " <<
average << endl;
    return 0;
}
```

Output

```
Enter the first number: 20
Enter the second number: 30
Enter the third number: 44
The average of 20, 30, and 44 is: 31.3333
```

Example 8.7:

What is a C++ program that prompts the user to enter the size of an array, then inputs a series of numbers into the array, calculates their average, and finally prints the average?

```
#include <iostream>
using namespace std;
// Function to calculate the average of elements in an array
double findAverage(double arr[], int size) {
    double sum = 0.0;
    // Calculate the sum of elements in the array
    for (int i = 0; i < size; i++) {
        sum += arr[i];
    }
    // Calculate the average
    return sum / size;
}
int main() {
    int size;
    // Ask the user for the size of the array
    cout << "Enter the size of the array: ";
    cin >> size;
    // Create an array to store the numbers
```

```
    double numbers[size];
    // Ask the user to input the elements of the array
    cout << "Enter " << size << " numbers:" << endl;
    for (int i = 0; i < size; i++) {
        cout << "Enter number " << i + 1 << ": ";
        cin >> numbers[i];
    }
    // Calculate the average
    double average = findAverage(numbers, size);
    // Print the result
    cout << "The average of the numbers is: " << average << endl;
    return 0;
}
```

Output

```
Enter the size of the array: 3
Enter 3 numbers:
Enter number 1: 20
Enter number 2: 30
Enter number 3: 44
The average of the numbers is: 31.3333
```

Example 8.8:

Write a C++ function to find the largest element with an array.

```
#include <iostream>
using namespace std;
// Function to find the largest element in an array
double findLargest(double arr[], int size) {
    // Assuming the first element as the largest initially
    double largest = arr[0];
    // Iterate through the array to find the largest element
    for (int i = 1; i < size; i++) {
```

```
      if (arr[i] > largest) {
         largest = arr[i];
      }
   }
   // Return the largest element
   return largest;
}
int main() {
   int size;
   // Ask the user for the size of the array
   cout << "Enter the size of the array: ";
   cin >> size;
   // Create an array to store the numbers
   double numbers[size];
   // Ask the user to input the elements of the array
   cout << "Enter " << size << " numbers:" << endl;
   for (int i = 0; i < size; i++) {
      cout << "Enter number " << i + 1 << ": ";
      cin >> numbers[i];
   }
   // Find the largest element
   double largest = findLargest(numbers, size);
   // Print the largest element
   cout << "The largest element in the array is: " << largest << endl;
   return 0;
}
```

Output

```
Enter the size of the array: 5
Enter 5 numbers:
Enter number 1: 20
Enter number 2: 30
```

```
Enter number 3: 50
Enter number 4: 21
Enter number 5: 33
The largest element in the array is: 50
```

Example 8.9:

Write a C++ program to find the factorial of each element on the diagonal of a square are of size n.

```
#include <iostream>
using namespace std;
// Function to calculate factorial
unsigned long long factorial(int n) {
   unsigned long long fact = 1;
   for (int i = 1; i <= n; i++) {
      fact *= i;
   }
   return fact;
}
int main() {
   int n;
   cout << "Enter the size of the square matrix: ";
   cin >> n;
   // Check if n is positive
   if (n <= 0) {
      cout << "Invalid input. Size of the matrix must be a positive integer." << endl;
      return 1;
   }
   // Creating a square matrix of size n
   int matrix[n][n];
   // Input the elements of the matrix
   cout << "Enter the elements of the matrix:" << endl;
   for (int i = 0; i < n; i++) {
```

```
    for (int j = 0; j < n; j++) {
        cin >> matrix[i][j];
    }
  }
  // Print the matrix
  cout << "Matrix:" << endl;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        cout << matrix[i][j] << " ";
    }
    cout << endl;
  }
  // Calculating factorial of each element on the diagonal
  cout << "Factorial of each element on the diagonal:" << endl;
  for (int i = 0; i < n; i++) {
    cout << "Element at position (" << i << ", " << i << "): ";
    cout << factorial(matrix[i][i]) << endl;
  }
  return 0;
}
```

Output

```
Enter the size of the square matrix: 3
Enter the elements of the matrix:
1
2
3
6
5
4
1
5
9
```

```
Matrix:
1 2 3
6 5 4
1 5 9
Factorial of each element on the diagonal:
Element at position (0, 0): 1
Element at position (1, 1): 120
Element at position (2, 2): 362880
```

3.6 Passing Array to a Function in C++ Programming

In C++, we can pass arrays as an argument to a function. And, also we can return arrays from a function.

The syntax for passing an array to a function is:

```
returnType functionName(dataType arrayName[]) {
   // code
}
```

Example 8.10:

This example demonstrates passing a one-dimensional array to a function. The program will display the marks of 5 students.

```
#include <iostream>
using namespace std;
const int SIZE = 5;
// declare a function to display marks, take a 1d array as parameter
void display(int m[]) {
   cout << "Displaying marks: " << endl;
   // display array elements
   for (int i = 0; i < SIZE; i++) {
      cout << "Student " << i + 1 << ": " << m[i] << endl;
   }
```

```
}
int main() {
   // declare and initialize an array
   int marks[SIZE] = {88, 76, 90, 61, 69};
   // call display function, pass an array as an argument
   display(marks);
   return 0;
}
```

Output

```
Displaying marks:
Student 1: 88
Student 2: 76
Student 3: 90
Student 4: 61
Student 5: 69
```

3.7 Passing Multidimensional Array to a Function

We can also pass multidimensional arrays as an argument to the function. For example,

Example 8.11:

This example demonstrates passing a multidimensional array to a function. The program displays the elements of a two-dimensional array by passing it to a function.

```
#include <iostream>
using namespace std;
// define a function, pass a 2d array as a parameter
void display(int n[][2]) {
   cout << "Displaying Values: " << endl;
   for (int i = 0; i < 3; i++) {
      for (int j = 0; j < 2; j++) {
         cout << "num[" << i << "][" << j << "]: " << n[i][j] << endl;
```

```
      }
   }
}
int main() {
   // initialize 2d array
   int num[3][2] = {
      {3, 4},
      {9, 5},
      {7, 1}
   };
   // call the function, pass a 2d array as an argument
   display(num);
   return 0;
}
```

Output

```
Displaying Values:
num[0][0]: 3
num[0][1]: 4
num[1][0]: 9
num[1][1]: 5
num[2][0]: 7
num[2][1]: 1
```

In the above program, we have defined a function named display(). The function takes a two-dimensional array, int n[][2] as its argument and prints the elements of the array. While calling the function, we only pass the name of the two-dimensional array as the function argument display(num).

Notes:

1) It is not mandatory to specify the number of rows in the array. However, the number of columns should always be specified. This is why we have used int n[][2].

2) We can also pass arrays with more than 2 dimensions as a function argument.
3) We can also return an array from the function. However, the actual array is not returned. Instead, the address of the first element of the array is returned with the help of **pointers**.

Example 8.12:

write a C++ program to add three 2d arrays of the same size. Use a sub-function for input, a sub-function for summation, and another one to print all the arrays and the sum.

–

Output

```
Input for Array A
Enter elements of the array:
Enter element at position [0][0]: 1
Enter element at position [0][1]: 2
Enter element at position [0][2]: 2
Enter element at position [1][0]: 5
Enter element at position [1][1]: 2
Enter element at position [1][2]: 1
Input for Array B
Enter elements of the array:
Enter element at position [0][0]: 2
Enter element at position [0][1]: 2
Enter element at position [0][2]: 1
Enter element at position [1][0]: 4
Enter element at position [1][1]: 2
Enter element at position [1][2]: 2
Input for Array C
Enter elements of the array:
Enter element at position [0][0]: 4
Enter element at position [0][1]: 5
Enter element at position [0][2]: 2
Enter element at position [1][0]: 1
Enter element at position [1][1]: 2
Enter element at position [1][2]: 4
Printing Arrays
---------------------
Array A:
```

```
1 2 2
5 2 1

Array B:
2 2 1
4 2 2

Array C:
4 5 2
1 2 4

Sum of Arrays:
7 9 5
10 6 7
```

3.8 Passing Structure to Function in C++

A structure variable can be passed to a function just like any other variable. Consider this example:

Example 8.13:

```
#include <iostream>
#include <string>
using namespace std;
struct Person {
   string first_name;
   string last_name;
   int age;
   float salary;
};
// declare a function with structure variable type as an argument
void display_data(const Person&);
int main() {
   // initialize the structure variable
   Person p {"John", "Doe", 22, 145000};
   // function call with structure variable as an argument
   display_data(p);
   return 0;
```

```
}
void display_data(const Person& p) {
   cout << "First Name: " << p.first_name << endl;
   cout << "Last Name: " << p.last_name << endl;
   cout << "Age: " << p.age << endl;
   cout << "Salary: " << p.salary;
}
```

Output

```
First Name: John
Last Name: Doe
Age: 22
Salary: 145000
```

In this program, we passed the structure variable **p** by reference to the function **display_data()** to display the members of **p**.

3.9 Return Structure From Function in C++

We can also return a structure variable from a function. Let's look at an example.

Example 8.14:

```
#include <iostream>
#include <string>
using namespace std;
// define structure
struct Person {
   string first_name;
   string last_name;
   int age;
   float salary;
};
// declare functions
Person get_data();
```

```
void display_data(const Person&);
int main() {
   Person p = get_data();
   display_data(p);
   return 0;
}
// define a function to return a structure variable
Person get_data() {
   Person p;
   string first_name;
   string last_name;
   int age;
   float salary;
   cout << "Enter first name: ";
   cin >> first_name;
   cout << "Enter last name: ";
   cin >> last_name;
   cout << "Enter age: ";
   cin >> age;
   cout << "Enter salary: ";
   cin >> salary;
   // return structure variable
   return Person{first_name, last_name, age, salary};
}
// define a function to take a structure variable as an argument
void display_data(const Person& p) {
   cout << "\nDisplaying Information." << endl;
   cout << "First Name: " << p.first_name << endl;
   cout << "Last Name: " << p.last_name << endl;
   cout << "Age: " << p.age << endl;
   cout << "Salary: " << p.salary;
}
```

Output

```
Enter first name: John
Enter last name: Doe
Enter age: 22
Enter salary: 145000

Displaying Information.
First Name: John
Last Name: Doe
Age: 22
Salary: 145000
```

In this program, we took user input for the structure variable in the `get_data()` function and returned it from the function. Then we passed the structure variable `p` to `display_data()` function by reference, which displays the information.

3.10 C++ Recursion

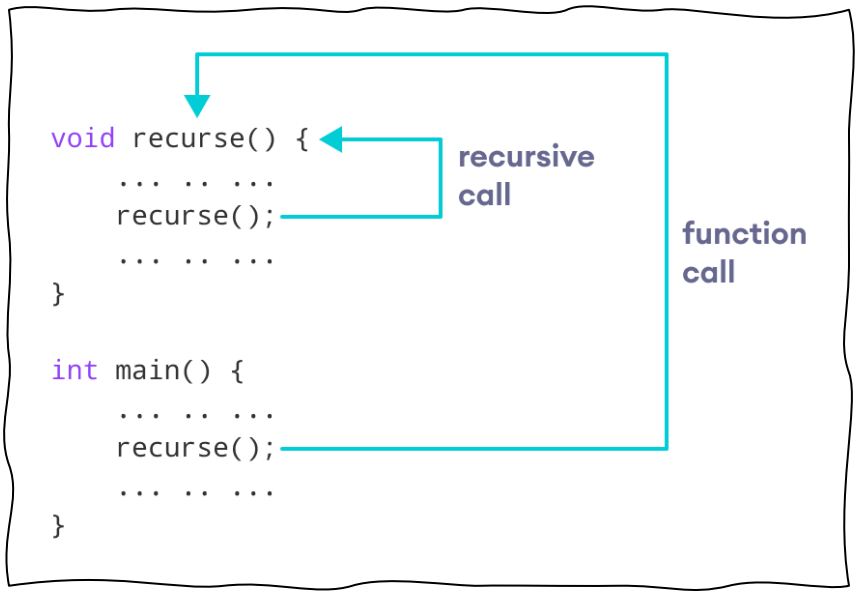
A function that calls itself is known as a recursive function. And, this technique is known as recursion.

```
void recurse()
{
   ... .. ...
   recurse();
   ... .. ...
}

int main()
{
```

```
    ... .. ...
    recurse();
    ... .. ...
}
```

The figure below shows how recursion works by calling itself over and over again.



How recursion works in C++ programming

The recursion continues until some condition is met.

To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call and the other doesn't.

Example 8.15:

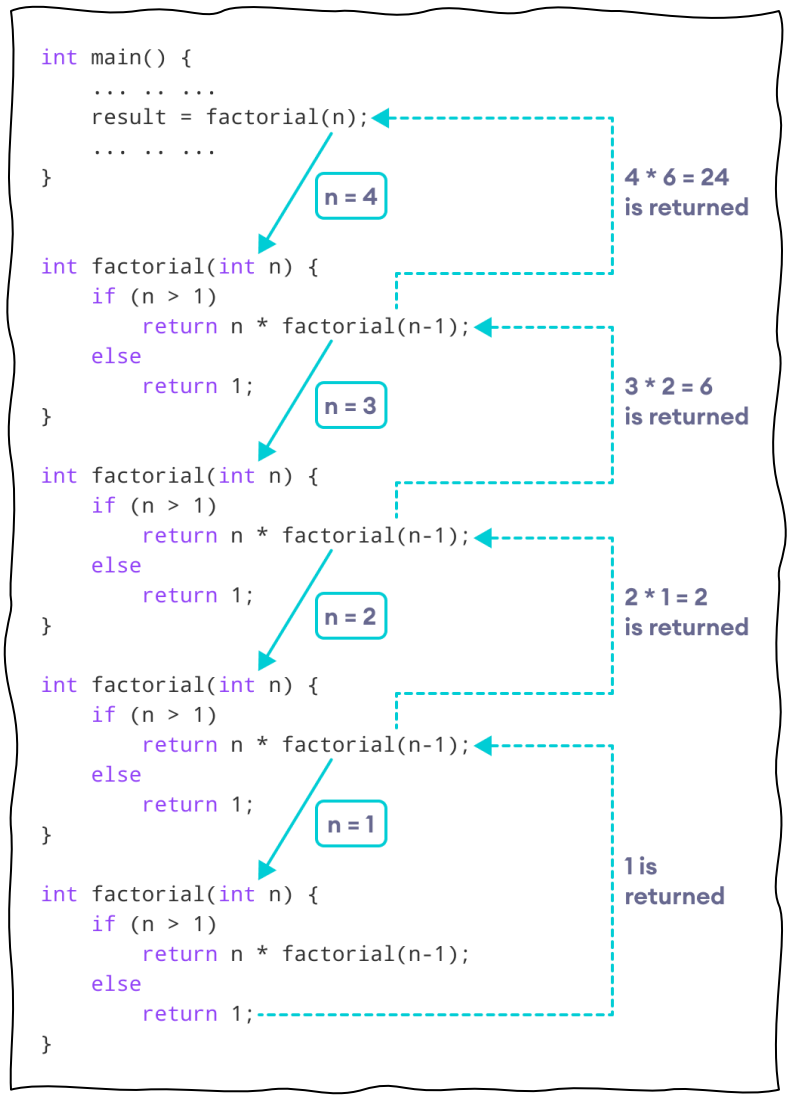
In this example, we will calculate the factorial of a number using recursion.

```
#include <iostream>
using namespace std;
int factorial(int);
int main() {
  int n, result;
  cout << "Enter a non-negative number: ";
  cin >> n;
```

```
    result = factorial(n);
    cout << "Factorial of " << n << " = " << result;
    return 0;
}
int factorial(int n) {
    if (n > 1) {
        return n * factorial(n - 1);
    } else {
        return 1;
    }
}
```

Output

```
Enter a non-negative number: 4
Factorial of 4 = 24
```



How this C++ recursion program works

Example 8.16:

Write a C++ program to compute the positive power of a number using recursion

```
#include <iostream>
using namespace std;
int calculatePower(int, int);
int main()
{
   int base, powerRaised, result;
   cout << "Enter base number: ";
```

```
    cin >> base;
    cout << "Enter power number(positive integer): ";
    cin >> powerRaised;
    result = calculatePower(base, powerRaised);
    cout << base << "^" << powerRaised << " = " << result;
    return 0;
}
int calculatePower(int base, int powerRaised)
{
    if (powerRaised != 0)
        return (base*calculatePower(base, powerRaised-1));
    else
        return 1;
}
```

Output

```
Enter base number: 3
Enter power number(positive integer): 4
3^4 = 81
```

This technique can only calculate power if the exponent is a positive integer.