OBJECT ORIENTED PROGRAMMING LAB

Lab Manual # 04

Functions and Arrays in C++

Instructor: Muhammad Abdullah Orakzai

Semester Fall, 2021

Course Code: CL1004

Fast National University of Computer and Emerging Sciences Peshawar

Department of Computer Science

OBJECT ORIENTED PROGRAMMING LANGUAGE

Table of Contents

Function

Function is a set of instructions that are designed to perform a specific task. A function is a complete and independent program. A function is a block of code which only runs when it is called. It is executed by the main function to perform its tasks. Functions are used to write the code of a large program by dividing it into smaller independent units. It avoids the replication of code in the program. A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions. You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task. A function is known with various names like a method or a sub-routine or a procedure etc.

Functions are like building blocks. They let you divide complicated programs into manageable pieces. They have other advantages, too:

1. While working on one function, you can focus on just that part of the program and construct it, debug it, and perfect it.
2. Different people can work on different functions simultaneously.
3. If a function is needed in more than one place in a program or in different programs, you can write it once and use it many times.
4. Using functions greatly enhances the program's readability because it reduces the complexity of the function main.

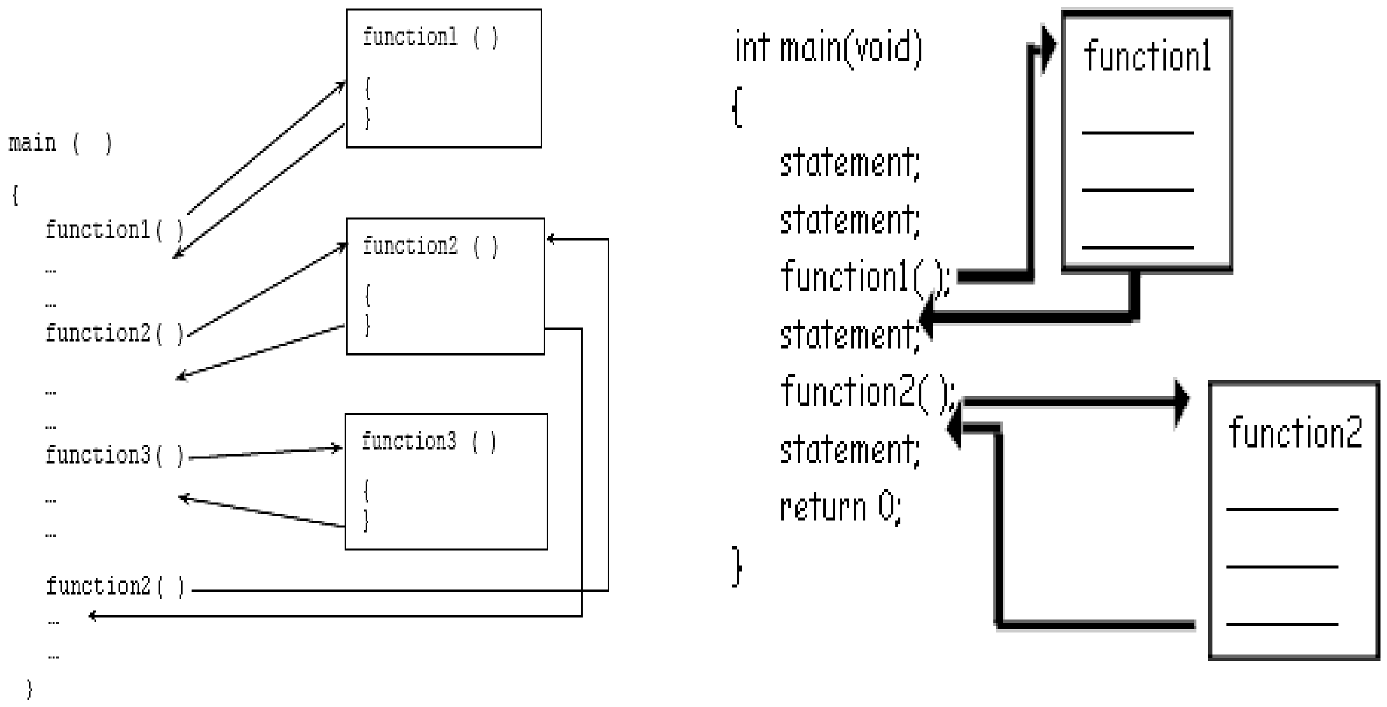
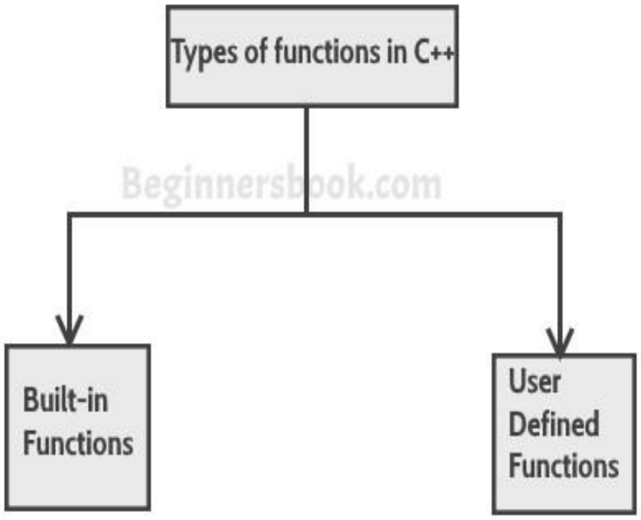


Figure 1: Functions

Types of Functions

1. Built in Functions or standard library functions

2. User Defined Functions



1. Built in Functions

The standard library methods are built-in functions in C++ that are readily available for use.

Built-in functions are also known as library functions. We need not to declare and define these functions as they are already written in the C++ libraries such as iostream, cmath etc. We can directly call them when we need.

For example pow(a,x), sqrt(x), round(x) etc.

2. User Defined Functions

❖ We can also create functions of our own choice to perform some task. Such functions are called user-defined functions.

❖ User define functions are the one that programmer writes it by himself.

```
void myFunction()
{

  cout<<"welcome to C++ Programming";

}
```

Defining a Function (Syntax)

```
return_type  function_name (parameter(s))
{
   //C++ Statements
}
```

Return Type – A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

Function Name – This is the actual name of the function. The function name and the parameter list together constitute the function signature.

Parameters – A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

Function Body – The function body contains a collection of statements that define what the function does.

Calling a or invoking function

- ❖ Executing the statement(s) of function to perform task is called calling a function.
- ❖ Calling a function is called invoking a function.
- ❖ Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are called.
- ❖ To call a function, write the function's name followed by two parentheses () and a semicolon ;

Example:

addition();

In the following example, myFunction() is used to print a text (the action), when it is called:

Example

Inside main, call myFunction():

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

// Create a function
void myFunction()
{
 cout<<"I just got executed!";
}

int main()
{

  myFunction(); // call the function
  return 0;
}

// Outputs "I just got executed!"
A function can be called multiple times:
Example
void myFunction() {

  cout << "I just got executed!\n";
}

int main() {
  myFunction();
  myFunction();
  myFunction();
  return 0;
}
/*
Output
// I just got executed!
// I just got executed!
// I just got executed!
*/
```

Function Declaration and Definition

A C++ function consist of two parts:

Declaration: the function's name, return type, and parameters (if any). Function declarations is also known prototype of the function.

Definition: the body of the function (code to be executed). The function definition consist of two parts:
(1) Declarator (2) Body of the function

void myFunction(int x, int v);

void **myFunction()** { // **declaration**
 // the body of the function (**definition**)
}

Note: If a user-defined function, such as myFunction() is declared after the main() function, **an error will occur**.

It is because C++ works from top to bottom; which means that if the function is not declared above main(), the program is unaware of it:

```
int main() {
  myFunction();
  return 0;
}

void myFunction() {
  cout << "I just got executed!";
}

// Error

/*
Output
In function 'int main()':
5:3: error: 'myFunction' was not declared in this scope
*/
```

- However, it is possible to separate the declaration and the definition of the function - for code optimization.
- You will often see C++ programs that have function declaration above main(), and function definition below main().
- This will make the code better organized and easier to read:

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

// Function declaration
void myFunction();

// The main method
int main() {
  myFunction();  // call the function
  return 0;
}

// Function definition
void myFunction() {
  cout << "I just got executed!";
}
/*
Output
I just got executed!
*/
```

Multiple Parameters

Inside the function, you can add as many parameters as you want:

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

void myFunction(string fname, int age) {
  cout << fname << "Khalil" << age << " years old. \n";
}

int main() {
  myFunction("Abid", 3);
  myFunction("Hassan", 14);
  myFunction("Yasin", 30);
  return 0;
}

/*
Output
Abid Khalil 3 years old.
Hassan Khalil 14 years old.
Yasin Khalil 30 years old.
*/
```

Functions different Scenarios

1. Function have no parameters list and return type.
2. Function have no return type but parameter list.
3. Function have both return type and parameter list (Function return values).
4. Function have return type but no parameter list.

1. Function have no parameters list and return type

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

void printStar()
{
    cout<<"**********";
}

int main()
{
  printStar(); // call the function
  return 0;
}
```

2. Function have no return type but parameter list

```
#include <iostream>
using namespace std;
void sum(int x, int y)  // parameters
    {
    int sum=x+y;
    cout<<"Result is: "<<sum;
    }
int main() {

    sum(5,6);      //Arguments
    return 0;
}
/*
Output
Result is: 11*/
```

Parameters or Arguments

- ❖ The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function.

From a function's perspective:

- ❖ A parameter is the variable listed inside the parentheses in the function definition.
- ❖ An argument is the value that is sent to the function when it is called.

3. Function have both return type and parameter list

Function return values

Function can return only one value.

Return Statement: The return statement is used to return calculated value from function definition to calling function.

Syntax:

return x;

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

int sum(int x, int y)  //parameters
    {
     return(x+y);
    }

int main() {

    int result=sum(5,6);      //Arguments
    cout<<"Result is: "<<result;
    //cout<<"Result is: "<<sum(5,6);
    return 0;
}

/*
Output
Result is: 11
*/
```

```
// function example
#include <iostream>
using namespace std;
int addition (int a, int b)
{
  int r;
  r=a+b;
  return r;
}

int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z<<endl;
}

/*
Output
The result is 8
*/
```

4. Function have return type but no parameter list

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

int sum()     {
    int x=50;
    int y=3;
    return(x+y);
    }

int main() {

    int result=sum();
    cout<<"Result is: "<<result;
    //cout<<"Result is: "<<sum();
    return 0;
}

/*
Output
Result is: 53
*/
```

Functions User Enters Arguments Values

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

int sum(int x, int y)  //parameters
    {
     return(x+y);
    }
```

```
int main() {
    int n1, n2;
    cout<<"Enter number 1:";
    cin>>n1;
    cout<<"Enter number 2:";
    cin>>n2;
    int result=sum(n1,n2);      //Arguments
    cout<<"Result is: "<<result;
    //cout<<"Result is: "<<sum(5,6);
    return 0;
}

/*
Output
Enter number 1: 3
Enter number 2: 3
Result is: 6
*/
```

How to pass arguments to functions

We can pass arguments to functions by two ways:

1. Passing by values.
2. Passing by references.

1. Pass by Values

Passing arguments in such a way where the function creates copies of the arguments passed to it , it is called passing by value.

When an argument is passed by value to a function, a new variable of the data type of the argument is created and the data is copied into it. The function accesses the value in the newly created variable and the data in the original variable in the calling function is not changed.

```
#include<iostream>
using namespace std;
void sum(int x, int y)
{
    int sum =x+y;
    cout<<"Result is: "<<sum;
}
int main()
{
    int a=5;
    int b=6;
    sum(a,b);
    return 0;
}
/*
Output
Result is: 11
*/
```

2. Pass by Reference

- The data can also be passed to a function by reference of a variable name and that contains data.
- The reference provides the second name (or **alias**) for a variable name.
- **Alias:** Two variables refer to the same thing or entity. Alias are the alternate name for referring to the same thing.
- When a variable is passed by reference to a function, no new copy of the variable is created. Only the address of the variable is passed to the function.
- The original variable is accessed in the function with reference to its second name or alias. Both variables use the same memory location.
- Thus, any change in the reference variable also changes the value in the original variable.
- The reference parameters are indicated by an ampersand (&) sign after the data type both in the function prototype and in the function definition.

```
#include <iostream>

using namespace std;
void swapNums(int &x, int &y) {
  int z = x;

  x = y;

  y = z;
}
int main() {

  int firstNum = 10;

  int secondNum = 20;
  cout << "Before swap: " << "\n";

  cout << firstNum << secondNum << "\n";
  swapNums(firstNum,  secondNum);
  cout << "After swap: " << "\n";

  cout << firstNum << secondNum << "\n";
  return 0;

}

/*
Output
Before swap:
10 20
After swap:
20 10
*/
```

- The ampersand sign (&) is also used with the data type of “x” and “y” variables. The x and y are the aliases of “firstNum” and “secondNum” variables respectively.
- The memory location of “x” and “firstNum” is the same and similarly, memory location of “y” and “secondNum” is same.

Function Example:

```
#include <iostream>
#include <cctype>
#include <iomanip>
using namespace std;
double getNumber();
char identifyInput();
double getKilos (double);
double getMeters (double);
```

```
int main()
{
double numberIn = 0.0, kilos = 0.0, meters = 0.0, answer = 0.0 ;
char choice;
numberIn = getNumber();
choice = identifyInput();
if (choice =='P')
{
answer = getKilos (numberIn);
cout << numberIn << " pounds are equivalent to " << answer << "
Kilograms.\n";
}
else
{
answer = getMeters (numberIn);
cout << numberIn << " yards are equivalent to " << answer << "
Meters.\n";
}
system("pause");
return 0;
}
double getNumber()
{
double input;
do
{
cout << "Enter a number greater than 0 that represents a measurement
in either pounds or yards: ";
cin >> input;
}while (input <= 0);
return input;
}
char identifyInput()
{
char option;
do
{
cout << "Enter a P if the number you input was pounds and Y if the
number input was yards: ";
cin >> option;
option = toupper(option);
}while(!(option == 'P' || option == 'Y'));
return option;
}
double getKilos (double lbs)
{
```

```
double result;
result = lbs * 0.45359237;
return result;
}
double getMeters ( double yds)
{
double result;
result = yds * 0.9144;
return result;
}
```

Output

```
/* Sample Run-I:
Enter a number greater than 0 that represents a measurement in either pounds or yards: 512
Enter a P if the number you input was pounds and Y if the number input was yards: P
512.00 pounds are equivalent to 232.24 Kilograms.

Sample Run-II:
Enter a number greater than 0 that represents a measurement in either pounds or yards: 512
Enter a P if the number you input was pounds and Y if the number input was yards: Y
512.00 yards are equivalent to 468.17 Meters.
Press any key to continue . . .
*/
```

Function Overloading in C++

Functions having same name with different set of parameters (type, order, number) then such kind of functions is called overloaded functions and this mechanism is called method overloading.

Function overloading is a compile time polymorphism or static binding. It increases the readability of the program.

Example

void myFunction(int x)
void myFunction(float x)
void myFunction(double x, double y)

Creating several functions with the same name but different formal parameters.
Two functions are said to have different formal parameter lists if both functions have:

- A different number of formal parameters or
- If the number of formal parameters is the same, then the data type of the formal parameters, in the order you list them, must differ in at least one position.

If a function's name is overloaded, then all of the functions in the set have the same name. Therefore, all of the functions in the set have different signatures if they have different formal parameter lists. Thus, the following function headings correctly overload the function functionXYZ:

```
void functionXYZ()
void functionXYZ(int x, double y)
void functionXYZ(double one, int y)
void functionXYZ(int x, double y, char ch)
```

Consider the following function headings to overload the function functionABC:

```
void functionABC(int x, double y)
int functionABC(int x, double y)
```

Both of these function headings have the same name and same formal parameter list. Therefore, these function headings to overload the function functionABC are incorrect. In this case, the compiler will generate a syntax error.

Consider the following example, which have two functions that add numbers of different type:

```
int plusFuncInt(int x, int y) {
  return x + y;
}

double plusFuncDouble(double x, double y) {
  return x + y;
}
int main() {
  int myNum1 = plusFuncInt(8, 5);

  double myNum2 = plusFuncDouble(4.3, 6.26);
  cout << "Int: " << myNum1 << "\n";
  cout << "Double: " << myNum2;
  return 0;
}
```

- Instead of defining two functions that should do the same thing, it is better to overload one.
- In the example below, we overload the plusFunc function to work for both int and double:

Function Overloading Example 1

```
#include <iostream>
using namespace std;
int plusFunc(int x, int y) {
  return x + y;
}

double plusFunc(double x, double y) {
  return x + y;
}

int main() {
  int myNum1 = plusFunc(8, 5);
  double myNum2 = plusFunc(4.3, 6.26);
  cout << "Int: " << myNum1 << "\n";
  cout << "Double: " << myNum2;
  return 0;
}

/*
Output
Int: 13
Double: 10.56
*/
```

Function Overloading Example 2

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

void sum(int x, int y)  // formal arguments
    {
    cout<<"sum of int is: "<<(x+y)<<endl;
    }
void sum(double x, double y)  // formal arguments
    {
    cout<<"sum of double is: "<<(x+y)<<endl;
    }
void sum(int x, double y)  // formal arguments
    {
    cout<<"sum of int & double is: "<<(x+y)<<endl;
    }
void sum(double x, int y)  // formal arguments
    {
    cout<<"sum of double & int is: "<<(x+y)<<endl;
    }

int main() {
    sum(3,5);
    sum(3.3,5.6);
    sum(3,5.4);
    sum(3.6,5);

  return 0;
}
/*
Output
sum of int is: 8
sum of double is: 8.9
sum of int & double is: 8.4
sum of double & int is: 8.6
*/
```

Function Overloading Example 3

```
#include <iostream>
#include <conio.h>
using namespace std;
void repchar();
void repchar(char);
void repchar(char, int);
void main()
{
repchar();
repchar('=');
repchar('+', 30);
system("pause");
}
void repchar()
{
for(int j=0; j<45; j++)
cout << '*';
cout << endl;
}
void repchar(char ch)
{
for(int j=0; j<45; j++)
cout << ch;
cout << endl;
}
```

```
void repchar(char ch, int n)
{
for(int j=0; j<n; j++)
cout << ch;
cout << endl;
}
```

Output

```
*******************************************
==========================================
+++++++++++++++++++++++++++++
```

Function with Default Parameters

- The default parameter is a way to set default values for function parameters a value is not passed in (i.e. it is undefined).
- In C++ programming, we can provide default values for function parameters.
- If a function with default arguments is called without passing arguments, then the default parameters are used.
- However, if arguments are passed while calling the function, the default arguments are ignored.
- You can also use a default parameter value, by using the equals sign (=).
- If we call the function without an argument, it uses the default value ("Norway"):

```
void myFunction(string country = "Norway")
{

 cout << country << "\n";
}


     myFunction("Sweden");

     myFunction();
```

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

void myFunction(string country = "Norway") {
  cout << country << "\n";
}

int main() {
  myFunction("Sweden");
  myFunction("India");
  myFunction();
  myFunction("USA");
  return 0;
}
/*
Output
Sweden
India
Norway
USA
*/
```

A parameter with a default value, is often known as an "optional parameter". From the example above, country is an optional parameter and "Norway" is the default value.

```
#include <iostream>
#include <conio.h>
using namespace std;
void repchar(char='*', int=45);
int main()
{
repchar();
repchar('=');
repchar('+', 30);
retun 0;
}
void repchar(char ch, int n)
{
for(int j=0; j<n; j++)
{cout << ch;}
}
/*
Output
*********************************************
=============================================
++++++++++++++++++++++++++++++
*/
```

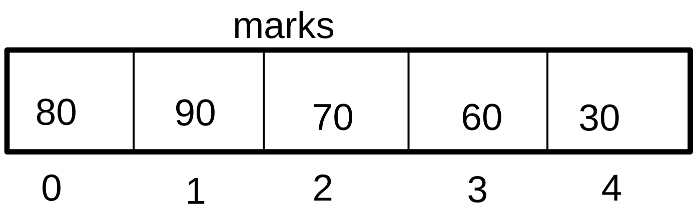
Arrays

- ❖ Same name which store multiple values. It is a collection of similar type of elements that have contiguous memory location.

Array is:

1. Linear data structure (consecutive location)
2. Static data structure (fixed size)
3. Homogeneous data will be stored.

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.



To declare an array, define the variable type, specify the name of the array followed by square brackets and specify the number of elements it should store:

To create an array of five integers, you could write:

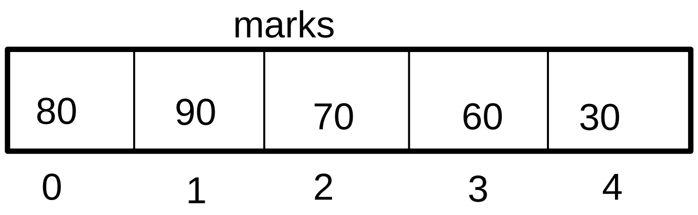
int myNum[5] = { 10, 20, 30, 40, 50 };

One Dimensional Array

A **one-dimensional array** is a structured collection of components (often called array elements) that can be accessed individually by specifying the position of a component with a single index value.

To create an array of five integers, you could write:

```
int marks[5] = {80, 90, 70, 60, 30};
```



Array Declaration

dataType arrayName[arraySize];

For example,

int x[6];

Here,

int - type of element to be stored

x - name of the array

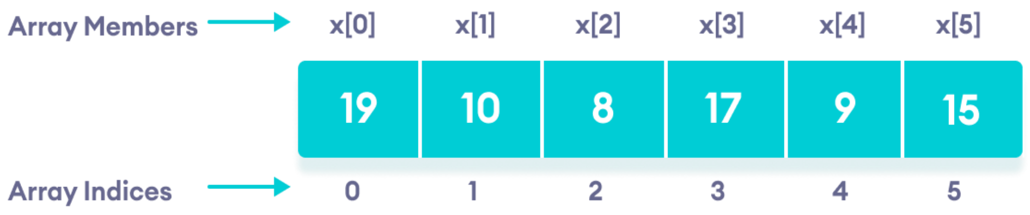
6 - size of the array

Array Initialization

In C++, it's possible to initialize an array during declaration. For example,

// declare and initialize and array

int x[6] = { 19, 10, 8, 17, 9, 15 };



Another method to initialize array during declaration:

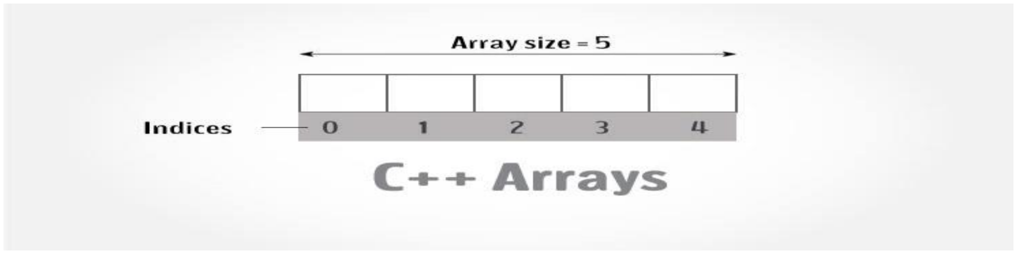
// declare and initialize an array

int x[] = {19, 10, 8, 17, 9, 15};

Here, we have not mentioned the size of the array. In such cases, the compiler automatically computes the size.

Access the Elements of an Array

To access the element you have to provide the index number along with the name.



cout << myNum[0];

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

int main() {

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

cout<<myNum[0]<<endl;
cout<<myNum[1]<<endl;
cout<<myNum[2]<<endl;
cout<<myNum[3]<<endl;
cout<<myNum[4]<<endl;

  return 0;
}

/*
Output
1
2
3
4
5
*/
```

Array with empty members

In C++, if an array has a size n, we can store upto n number of elements in the array. However, what will happen if we store less than n number of elements.

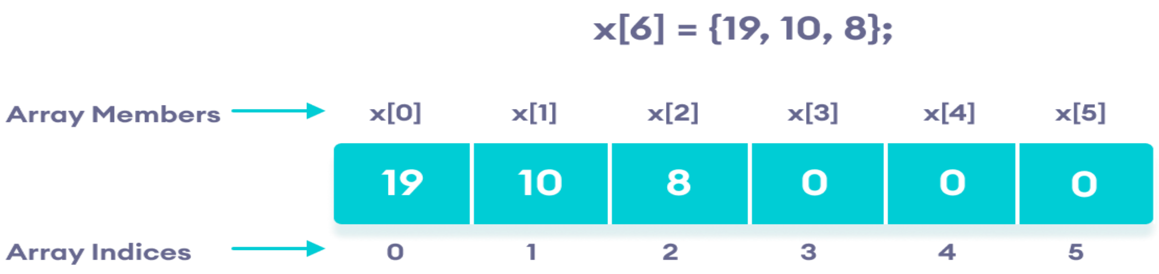
For example,

// store only 3 elements in the array

int x[6] = {19, 10, 8};

Here, the array x has a size of 6. However, we have initialized it with only 3 elements.

In such cases, the compiler assigns random values to the remaining places. Oftentimes, this random value is simply 0.



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

int main() {

int myNum[6] ={1,2,3};

for(int number: myNum)
{
    cout<<number<<endl;
}


  return 0;

}

/*
Output
1
2
3
0
0
0
*/
```

Change an Array Element

To change the value of a specific element, refer to the index number:

Example

myNum [2] = 4000;

```
#include<iostream>
using namespace std;
int main() {
int myNum[] ={1,2,3,4,5,6,7};
cout<<"Value at myNum[2]: "<<myNum[2]<<endl;

myNum[2]=7777;

cout<<"Value at myNum[2]: "<<myNum[2];

  return 0;
}
/*
Output
Value at myNum[2]: 3
Value at myNum[2]: 7777
*/
```

Loop through an Array (for loop)

You can loop through the array elements with the for loop.

The following example outputs all elements in the **myNum** array:

int myNum[5] = {10, 20, 30, 40, 50};

for(int i = 0; i < 4; i++)

{
 cout <<myNum[i] <<endl;
}

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

int main() {
int myNum[] ={1,2,3,4,5,6,7};

cout<<"***Array Iteration Using for loop***"<<endl;
for(int i=0; i<sizeof(myNum)/sizeof(int); i++)
{
cout<<myNum[i]<<endl;

}
  return 0;
}
/*
Output
1
2
3
4
5
6
7
*/
```

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

int main() {
int myNum[] ={1,2,3,4,5,6,7};

cout<<"***Array Iteration Using for loop***"<<endl;
for(int i=0; i<7; i++)
{

cout<<myNum[i]<<endl;

}
  return 0;
}
/*
Output
1
2
3
4
5
6
7
*/
```

Loop through an Array (enhanced for loop)

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

int main() {
int myNum[] ={1,2,3,4,5,6,7};

cout<<"***Array Iteration Using enhanced for loop***"<<endl;

for (int number : myNum)
{
    cout<<number<<endl;
}
  return 0;
}

/*
Output
***Array Iteration Using enhanced for loop***
1
2
3
4
5
6
7
*/
```

Advantages of an Array in C/C++

- Random access of elements using array index.
- Use of less line of code as it creates a single array of multiple elements.
- Easy access to all the elements.
- Traversal through the array becomes easy using a single loop.
- Sorting becomes easy as it can be accomplished by writing less line of code.

Disadvantages of an Array in C/C++

- Allows a fixed number of elements to be entered which is decided at the time of declaration. Unlike a linked list, an array in C is not dynamic.

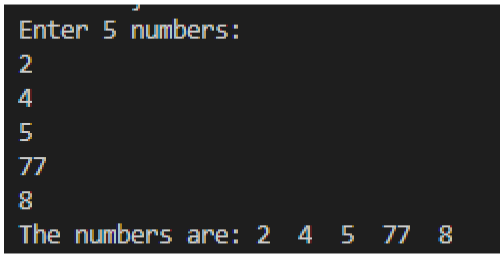
- ❖ Insertion and deletion of elements can be costly since the elements are needed to be managed in accordance with the new memory allocation.

Take Inputs from User and Store Them in an Array

```
#include <iostream>
using namespace std;
int main() {
    int numbers[5];
    cout << "Enter 5 numbers: " << endl;

    //  store input from user to array
    for (int i = 0; i < 5; ++i) {
        cin >> numbers[i];
    }
    cout << "The numbers are: ";

    //  print array elements
    for (int n = 0; n < 5; ++n) {
        cout << numbers[n] << "  ";
    }
    return 0;
}
```



```
Enter 5 numbers:
2
4
5
77
8
The numbers are: 2  4  5  77  8
```

- ❖ Once again, we have used a for loop to iterate from i = 0 to i = 4. In each iteration, we took an input from the user and stored it in numbers[i].
- ❖ Then, we used another for loop to print all the array elements.

String Arrays

```
#include<iostream>
using namespace std;
int main() {
string names[4] = {"Ali", "Asia", "Zain", "Zainab"};
for(int i = 0; i < 4; i++)
{
  cout << names[i] << "\n";
}
  return 0;
}

/*
Output
Ali
Asia
Zain
Zainab
*/
```

The following example outputs the index of each element together with its value:

```
#include<iostream>
using namespace std;
int main() {
string names[4] = {"Ali", "Asia", "Zain", "Zainab"};
for(int i = 0; i < 4; i++) {
cout << i << ": " << names[i] << "\n";

}
  return 0;
}
/*
Output
0: Ali
1: Asia
2: Zain
3: Zainab
*/
```

How to Pass 1D array to function

```
#include<iostream>
using namespace std;
void arrayIterationFunction(int test[])
{
    for (int i = 0; i <6 ; i++)
    {
        cout<<"myNum["<<i<<"]="<<test[i]<<endl;
    }
}
int main()
{
    int myNum[]= {1,3,4,5,6,7};
    arrayIterationFunction(myNum);
}
/*
Output
myNum[0]=1
myNum[1]=3
myNum[2]=4
myNum[3]=5
myNum[4]=6
myNum[5]=7
*/
```

```
#include<iostream>
using namespace std;
int array_size;    // initialization at the top
void arrayIterationFunction(int test[])
{
    for (size_t i = 0; i <array_size; i++)
    {
        cout<<"Enter value at test["<<i<<"]=";
        cin>>test[i];
    }

    for (int i = 0; i <array_size; i++)
    {
        cout<<"myNum["<<i<<"]="<<test[i]<<endl;
    }
}
int main()
{
    cout<<"Enter Array Size: ";
    cin>>array_size;
    int myNum[array_size];

    arrayIterationFunction(myNum);
}
/*
Output
Enter Array Size: 4
Enter value at test[0]=3
Enter value at test[1]=2
Enter value at test[2]=1
Enter value at test[3]=6
myNum[0]=3
myNum[1]=2
myNum[2]=1
myNum[3]=6
*/
```

Find Maximum Value in 1 D Array

```
#include <iostream>

using namespace std;
int main() {
int array_size= 5;

int myNum[array_size]= {11,10,33,4,5};

int max =myNum[0];


for (int i = 0; i <array_size; i++)

{

    if (myNum[i] > max)

    {

        max=myNum[i];

    }

}

cout<<"Maximum number is: "<<max;


  return 0;

}


Output:

Maximum number is: 33
```

Find Minimum Value in 1 D Array

```
#include <iostream>

using namespace std;
int main() {


int array_size= 5;

int myNum[array_size]= {11,10,33,4,5};

int min =myNum[0];


for (int i = 0; i <array_size; i++)

{

    if (myNum[i] < min)

    {

        min=myNum[i];

    }


}

cout<<"Manimum number is: "<<min;
  return 0;

}
```

Output:

```
Manimum number is: 4
```

Multidimensional Array

- ❖ C++ allows multidimensional arrays. Here is the general form of a multidimensional array declaration –
- ❖ type name[size1] [size2]...[sizeN];
- ❖ For example, the following declaration creates a three dimensional 5 . 10 . 4 integer array –

  - int threedim[5][10][4];

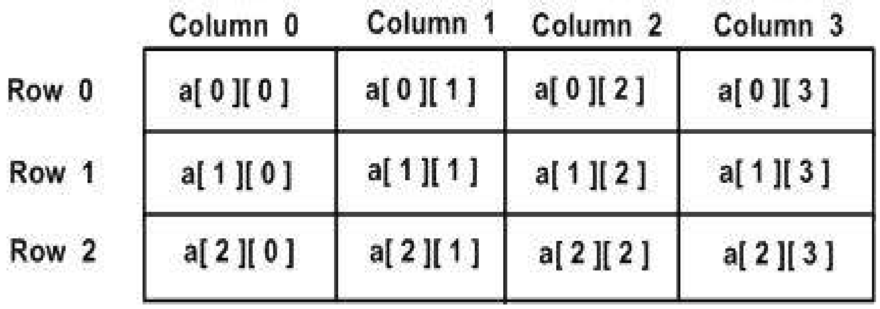
❖ But our focus will on 2- dimensional arrays

Two Dimensional Arrays (2D Array)

- The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays.
- An array that is represented with two indices/subscripts is called 2D array.
- It is similar to matrix in maths.
- Logically it consists of rows and columns.
- 2D array is called an array of an arrays.
- To declare a two-dimensional integer array of size x,y, you would write something as follows:

  type arrayName [r][c];

- Where **type** can be any valid C++ data type and **arrayName** will be a valid C++ identifier.
- A two-dimensional array can be think as a table, which will have **r** number of rows and **c** number of columns.
- A 2-dimensional array **a**, which contains three rows and four columns can be shown as below:



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in array a is identified by an element name of the form **a[r][c]**, where **a** is the name of the array, and **r** and **c** are the subscripts that uniquely identify each element in **a**.

2D Array logical Representation

	0	1	2
0	**(0,0** 7	**(0,1** 8	**(0,2** 9
1	**(1,0** 1	**(1,1** 2	**(1,2** 3
2	**(2,0** 5	**(2,1** 1	**(2,2** 1
3	**(3,0** 5	**(3,1** 6	**(3,2** 7

Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row have 4 columns.

```
int a[3][4] = {    {0, 1, 2, 3} ,      /*  initializers for row indexed by 0 */

                   {4, 5, 6, 7} ,      /*  initializers for row indexed by 1 */

                   {8, 9, 10, 11}      /*  initializers for row indexed by 2 */

              } ;

int a[3][4] = {    {0, 1, 2, 3}    ,    {4, 5, 6, 7}    ,   {8, 9, 10, 11}  };
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example –

```
int a[3][4] = {0,1,2,3,   4,5,6,7,8,    9,10,11};
```

Accessing Two-Dimensional Array Elements

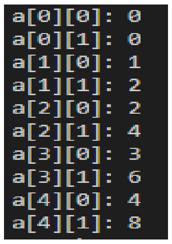
An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example −

```
int val = a[2][3];     //assigning value of a[2][3] that is 11 to variable val

cout<<a[2][3];
```

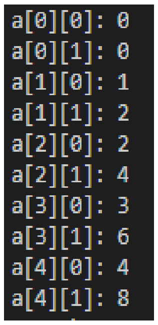
The above statement will take 4th element from the 3rd row of the array. You can verify it in the above diagram.

```
#include <iostream>
using namespace std;
int main () {
   // an array with 5 rows and 2 columns.
   int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
   // output each array element's value
   for ( int r = 0; r < 5; r++ )
     {
           for ( int c = 0; c < 2; c++ )
     {
            cout << "a[" << r << "][" << c << "]: ";
            cout << a[r][c]<< endl;
           }
     }
   return 0;
}
```



```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

❖ When the above code is compiled and executed, it produces the following result –



```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

❖ As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

Inline Functions

The inline function is special function used for small functions. It is a user-defined function but in it the function prototype is omitted. The keyword "**inline**" is used in the declarator on the function definition. The "**inline**" function is defined before the main function.

When a compiler compiles the source code, it generates a special code at the function call to jump to the function definition. It again generates a code at the end of the function definition to jump back to the calling function. Thus, time is spent is passing the control to the function body and then to the calling function during execution of the program.

The code of the function can be inserted at each of the function call to save the jumping time during program execution. By using the keyword "**inline**" in the declarator of the function definition, the code of the function body is inserted at the place of function call during compilation. Such types of functions are called inline functions.

Note: The inline functions are normally used for small functions.

Inline Functions Example

Write a program to find the exponential power of a given number.

```
#include <iostream>

using namespace std;
inline int expp(int b, int e)

{

    int c;

    int pp=1;

    for(c=1; c<=e; c++)

    {

        pp*=b;  // pp=pp*b;

    }

    return pp;

}
```

```
int main()
{
    int n, p, res;
    cout<<"Enter any number?"<<endl;
    cin>>n;
    cout<<"Enter raise to the power?"<<endl;
    cin>>p;

    res= expp(n, p);
    cout<<"The exponential power of given number is ="<<res;

    return 0;
}
```

Output:

```
Enter any number?
2
Enter raise to the power?
3
The exponential power of given number is =8
```

C++ Templates

In this topic, you'll learn about templates in C++. You'll learn to use the power of templates for generic programming.

Templates are powerful features of C++ which allows you to write generic programs. In simple terms, you can create a single function or a class to work with different data types using templates.

Templates are often used in larger codebase for the purpose of code reusability and flexibility of the programs.

The concept of templates can be used in two different ways:

- Function Templates
- Class Templates

Function Templates

A function template works in a similar to a normal function, with one key difference.

A **single function template** can work with different data types at once but, a **single normal function** can only work with one set of data types.

Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.

However, a better approach would be to use function templates because you can perform the same task writing less and maintainable code.

In function overloading more than one function with the same name are declared and defined. This makes the program more complex and lengthy. To overcome this problem, function template is used.

Function Template Declaration

In a function template a single typeless function is defined such that arguments of any standard data type can be passed to the function.

The general syntax of a function template is:

```
template <class T>

T function_name(T  argument(s))

{

  Statement(s);

}
```

template	The keyword template is used to define the function template.
class	In function templates a general data name is defined by using the keyword "**class**". In the above syntax, it is represented by word "T".

Any character or word can be used after the keyword "**class**" to specify the general data name.

function_name It specifies the name of the function.

argument(s) These are the arguments that are to be passed to the function. The arguments are declared of type **T**. As explained earlier, the type **T** specifies any standard data type.

statement(s) It specifies the actual statements of the function.

For example, to write a function to calculate a sum of three values either integers or float-points, the function template is written as:

```
template <class T>

T sum(T a, T b, T c)

{

  return (a+b+c);

}
```

Example 1: Function Template to sum of three values

Write a program to calculate and print the sum of three integer and floating-point values. Use the function template.

```
#include <iostream>

//Program to swap data using function templates.

#include <iostream>

using namespace std;


template <class T>

T sum(T a, T b, T c)

{

   return (a+b+c);

}
```

```
int main()
{
    cout<<"Sum of 3 integer values ="<<sum(33,44,55)<<endl;
    cout<<"Sum of 3 float values ="<<sum(33.5,44.55,55.66)<<endl;
    return 0;
}
```

Output:

```
Sum of 3 integer values =132
Sum of 3 float values =133
```

Note: In template functions, the function prototype is generally omitted.

Example 2: Function Template to find the largest number

//Program to display largest value among two numbers using function templates.

```
// If two characters are passed to function template, character with larger ASCII
value is displayed.
#include <iostream>
using namespace std;
// template function
template <class T>
T Large(T n1, T n2)
{
        return (n1 > n2) ? n1 : n2;
}

int main()
{
        int i1, i2;
        float f1, f2;
        char c1, c2;
        cout << "Enter two integers:\n";
        cin >> i1 >> i2;
        cout << Large(i1, i2) <<" is larger." << endl;

        cout << "\nEnter two floating-point numbers:\n";
        cin >> f1 >> f2;
        cout << Large(f1, f2) <<" is larger." << endl;

        cout << "\nEnter two characters:\n";
        cin >> c1 >> c2;
        cout << Large(c1, c2) << " has larger ASCII value.";
        return 0;
}
```

Output

```
Enter two integers:
5
10
10 is larger.

Enter two floating-point numbers:
12.4
10.2
12.4 is larger.

Enter two characters:
z
Z
z has larger ASCII value.
```

- In the above program, a function template Large() is defined that accepts two arguments n1 and n2 of data type T. T signifies that argument can be of any data type.
- Large() function returns the largest among the two arguments using a simple conditional operation.
- Inside the main() function, variables of three different data types: int, float and char are declared. The variables are then passed to the Large() function template as normal functions.
- During run-time, when an integer is passed to the template function, compiler knows it has to generate a Large() function to accept the int arguments and does so.
- Similarly, when floating-point data and char data are passed, it knows the argument data types and generates the Large() function accordingly.
- This way, using only a single function template replaced three identical normal functions and made your code maintainable.

Example 3: Swap Data Using Function Templates

//Program to swap data using function templates.

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

template <typename T>
void Swap(T &n1, T &n2)
{
        T temp;
        temp = n1;
        n1 = n2;
        n2 = temp;
}

int main()
{
        int i1 = 1, i2 = 2;
        float f1 = 1.1, f2 = 2.2;
        char c1 = 'a', c2 = 'b';

        cout << "Before passing data to function template.\n";
        cout << "i1 = " << i1 << "\ni2 = " << i2;
        cout << "\nf1 = " << f1 << "\nf2 = " << f2;
        cout << "\nc1 = " << c1 << "\nc2 = " << c2;

        Swap(i1, i2);
        Swap(f1, f2);
        Swap(c1, c2);

        cout << "\n\nAfter passing data to function template.\n";
        cout << "i1 = " << i1 << "\ni2 = " << i2;
        cout << "\nf1 = " << f1 << "\nf2 = " << f2;
        cout << "\nc1 = " << c1 << "\nc2 = " << c2;

        return 0;
}
```

Output

```
Before passing data to function template.
```

```
i1 = 1

i2 = 2

f1 = 1.1

f2 = 2.2

c1 = a

c2 = b

After passing data to function template.

i1 = 2

i2 = 1

f1 = 2.2

f2 = 1.1

c1 = b

c2 = a
```

In this program, instead of calling a function by passing a value, a call by reference is issued.

The Swap() function template takes two arguments and swaps them by reference.

References

https://beginnersbook.com/2017/08/cpp-data-types/

http://www.cplusplus.com/doc/tutorial/basic_io/

https://www.w3schools.com/cpp/default.asp

https://www.javatpoint.com/cpp-tutorial

https://www.geeksforgeeks.org/object-oriented-programming-in-cpp/?ref=lbp

https://www.programiz.com/

https://ecomputernotes.com/cpp/