

UNIT- I

C++ Programming Concepts: Review of C, input and output in C++, functions in C++- value parameters, reference parameters, Parameter passing, function overloading, function templates, arrays, pointers, new and delete operators, class and object, access specifiers, friend functions, constructors and destructor, Operator overloading, class templates.

Procedural Programming vs Object-Oriented Programming

Below are some of the differences between procedural and object-oriented programming:

- **Procedural Oriented Programming**

- In procedural programming, the program is divided into small parts called *functions*.
- Procedural programming follows a *top-down approach*.
- There is no access specifier in procedural programming.
- Adding new data and functions is not easy.

- **Object-Oriented Programming**

- In object-oriented programming, the program is divided into small parts called *objects*.
- Object-oriented programming follows a *bottom-up approach*.
- Object-oriented programming has access specifiers like private, public, protected, etc.
- Adding new data and function is easy.

- Procedural programming does not have any proper way of hiding data so it is *less secure*.
- In procedural programming, overloading is not possible.
- In procedural programming, there is no concept of data hiding and inheritance.
- In procedural programming, the function is more important than the data.
- Procedural programming is based on the *unreal world*.
- Procedural programming is used for designing medium-sized programs.
- Procedural programming uses the concept of procedure abstraction.
- Object-oriented programming provides data hiding so it is *more secure*.
- Overloading is possible in object-oriented programming.
- In object-oriented programming, the concept of data hiding and inheritance is used.
- In object-oriented programming, data is more important than function.
- Object-oriented programming is based on the *real world*.
- Object-oriented programming is used for designing large and complex programs.
- Object-oriented programming uses the concept of data abstraction.

- Code reusability absent in procedural programming,
- Examples: C, FORTRAN, Pascal, Basic, etc.
- Code reusability present in object-oriented programming.
- Examples: C++, Java, Python, C#, etc.

C++ Basic Input/Output

C++ I/O operation is using the stream concept. Stream is the sequence of bytes or flow of data. It makes the performance fast.

`<iostream>`

It is used to define the cout, cin and cerr objects, which correspond to standard output stream, standard input stream and standard error stream, respectively.

Standard output stream (cout)

The cout is a predefined object of ostream class. It is connected with the standard output device, which is usually a display screen. The cout is used in conjunction with stream insertion operator (<<) to display the output on a console.

```
#include <iostream>
```

```
using namespace std;
```

```
int main( )
```

```
{
```

```
    cout << "Welcome to CPP " << endl;
```

```
}
```

Standard input stream (cin)

The cin is a predefined object of istream class. It is connected with the standard input device, which is usually a keyboard. The cin is used in conjunction with stream extraction operator (>>) to read the input from a console.

```
#include <iostream>
```

```
using namespace std;

int main( ) {

    int age;

    cout << "Enter your age: ";

    cin >> age;

    cout << "Your age is: " << age << endl;

}
```

Standard end line (endl)

The endl is a predefined object of ostream class. It is used to insert a new line characters and flushes the stream.

```
#include <iostream>

using namespace std;

int main( ) {

    cout << "C++ Tutorial";

    cout << " Javatpoint"<<endl;

    cout << "End of line"<<endl;

}
```

Functions in CPP

In programming, function refers to a segment that groups code to perform a specific task.

Depending on whether a function is predefined or created by programmer; there are two types of function:

1. Library Function
2. User-defined Function

Library Function:

Library functions are the built-in function in C++ programming. Programmer can use library function by invoking function directly; they don't need to write it themselves.

Example program on usage of library functions

```
#include <iostream>
```

```
#include <cmath>

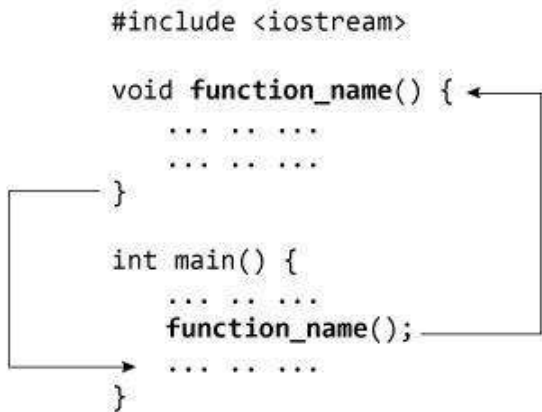
using namespace std;

int main()
{
    double number, squareRoot;
    cout << "Enter a number: ";
    cin >> number;

    // sqrt() is a library function to calculate square root
    squareRoot = sqrt(number);
    cout << "Square root of " << number << " = " << squareRoot;
    return 0;
}
Enter a number: 26
Square root of 26 = 5.09902
```

User-defined Function

C++ allows 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 invoked from any part of program, it all executes the codes defined in the body of function.



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

// Function prototype (declaration)
int add(int, int);

int main()
{
    int num1, num2, sum;
    cout<<"Enters two numbers to add: ";
```

```

cin >> num1 >> num2;

// Function call
sum = add(num1, num2);
cout << "Sum = " << sum;
return 0;
}

// Function definition
int add(int a, int b)
{
    int add;
    add = a + b;

    // Return statement
    return add;
}

```

Passing Arguments to Function

In programming, argument (parameter) refers to the data which is passed to a function (function definition) while calling it.

In the above example, two variables, num1 and num2 are passed to function during function call. These arguments are known as actual arguments.

The value of num1 and num2 are initialized to variables a and b respectively. These arguments a and b are called formal arguments.

```

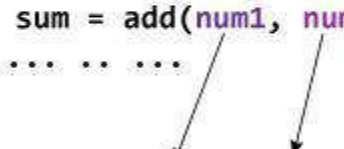
# include <iostream>
using namespace std;

int add(int, int);

int main() {
    ... ..
    sum = add(num1, num2); // Actual parameters: num1 and num2
    ... ..
}

int add(int a, int b) { // Formal parameters: a and b
    ... ..
    add = a+b;
    ... ..
}

```



Parameter passing

There are different ways in which parameter data can be passed into and out of methods and functions. Let us assume that a function B() is called from another function A(). In this case A is called the “caller function” and B is called the “called function or callee function”. Also, the arguments which A sends to B are called actual arguments and the parameters of B are called formal arguments.

Important methods of Parameter Passing

Call By Value:

Changes made to formal parameter do not get transmitted back to the caller. Any modifications to the formal parameter variable inside the called function or method affect only the separate storage location and will not be reflected in the actual parameter in the calling environment. This method is also called as pass by value.

```
#include <iostream>

using namespace std;

void swap(int x, int y);

void swap(int x, int y) {
    int temp;
    temp = x;
    x = y;
    y = temp;
}

int main ()
{
    int a = 100;
    int b = 200;
    cout << "Before swap, value of a :" << a << endl;
    cout << "Before swap, value of b :" << b << endl;
    swap(a, b);
    cout << "After swap, value of a :" << a << endl;
    cout << "After swap, value of b :" << b << endl;
    return 0;
}
```

Call by reference

Changes made to formal parameter do get transmitted back to the caller through parameter passing. Any changes to the formal parameter are reflected in the actual parameter in the calling environment as formal parameter receives a reference (or pointer) to the actual data. This method is also called as pass by reference.

```
#include <iostream>

using namespace std;

void swap(int &x, int &y);

void swap(int &x, int &y) {
    int temp;
    temp = x;
    x = y;
    y = temp;
}

int main ()
{
    int a = 100;
    int b = 200;

    cout << "Before swap, value of a :" << a << endl;
    cout << "Before swap, value of b :" << b << endl;
    swap(a, b);
    cout << "After swap, value of a :" << a << endl;
    cout << "After swap, value of b :" << b << endl;
    return 0;
}
```

Call by pointer

The call by pointer method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

```
#include <iostream>

using namespace std;
```



```

void swap(int *x, int *y);
void swap(int *x, int *y) {
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
int main () {
    int a = 100;
    int b = 200;
    cout << "Before swap, value of a :" << a << endl;
    cout << "Before swap, value of b :" << b << endl;
    swap(&a, &b);
    cout << "After swap, value of a :" << a << endl;
    cout << "After swap, value of b :" << b << endl;
    return 0;
}

```

C++ Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.

```

#include <iostream>
using namespace std;
class Cal {
    public:
    static int add(int a,int b){
        return a + b;
    }
    static int add(int a, int b, int c)
    {

```

```

        return a + b + c;
    }
};
int main(void) {
    Cal C;                                // class object declaration.
    cout<<C.add(10, 20)<<endl;
    cout<<C.add(12, 20, 23);
    return 0;
}

```

Function templates

Are **special functions that can operate with generic types**. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

```

template <typename T>
T functionName(T parameter1, T parameter2, ...)
{
    // code
}

```

Example

```

int main() {

    int result1;
    double result2;

    // calling with int parameters
    result1 = add<int>(2, 3);
    cout << result1 << endl;

    // calling with double parameters
    result2 = add<double>(2.2, 3.3);
    cout << result2 << endl;

    return 0;
}

```

C++ array

The array is a user-defined data type which is a collection of similar data types, such as a set of integers or characters. we cannot store different data type in the array at a time. Where in the value is stored as a series or sequence.

Difference between array and normal variable in C++

As we know, one variable can store one value at the same time, while using the array you can store a lot of value (which are of the same type) in a variable at the same time.

for example,

using a normal variable

```
int x=5;
```

while using array variable

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

Type of array in C++

- single dimensional array
- multi-Dimensional array

Single Dimensional Array

In this type of array, it stores elements in a single dimension. And, In this array, a single specification is required to describe elements of the array.

```
#include <iostream>
using namespace std;
int main()
{
    int arr[5]={10, 0, 20, 0, 30}; //creating and initializing array
    //traversing array
    for (int i = 0; i < 5; i++)
    {
        cout<<arr[i]<<"\n";
    }
}
```

Two-Dimensional Array:

In this type of array, two indexes describe each element, the first index represents a row, and the second index represents a column.

```
#include <iostream>

using namespace std;

int main()
{
    // a 2x3 array
    int a[3][2] = { {0, 2}, {1, 4}, {3, 7} };

    // traverse array elements

    for (int i=0; i<3; i++)
        for (int j=0; j<2; j++)
        {
            cout << "a[" <<i<< "]" <<j<< "]: ";
            cout << a[i][j] << endl;
        }
    return 0;
}
```

Pointers

A **pointer** is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is –

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C++ type and **var-name** is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration –

```
int *ip; // pointer to an integer
double *dp; // pointer to a double
```

Using Pointers in C++

There are few important operations, which we will do with the pointers very frequently. **(a)** We define a pointer variable. **(b)** Assign the address of a variable to a pointer. **(c)** Finally access the value at the address available in the pointer variable. This is done by using unary operator ***** that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations –

```
#include <iostream>

using namespace std;

int main () {
    int var = 20; // actual variable declaration.
    int *ip;      // pointer variable

    ip = &var;    // store address of var in pointer variable

    cout << "Value of var variable: ";
    cout << var << endl;

    // print the address stored in ip pointer variable
    cout << "Address stored in ip variable: ";
    cout << ip << endl;

    // access the value at the address available in pointer
    cout << "Value of *ip variable: ";
    cout << *ip << endl;

    return 0;
}
```

Dynamic memory allocation in C++

There are times where the data to be entered is allocated at the time of execution. For example, a list of employees increases as the new employees are hired in the organization and similarly reduces when a person leaves the organization. This is called managing the memory. So now, let us discuss the concept of dynamic memory allocation.

Dynamic allocation or run-time allocation - The allocation in which memory is allocated dynamically. In this type of allocation, the exact size of the variable is not known in advance. Pointers play a major role in dynamic memory allocation.

Dynamic memory allocation using the new operator

To allocate the space dynamically, the operator new is used. It means creating a request for memory allocation on the free store. If memory is available, memory is initialized, and the address of that space is returned to a pointer variable.

Syntax

```
Pointer_variable = new data_type;
```

```
int *m = new int(20);
```

```
float *d = new float(21.01);
```

We can also use a new operator to allocate a block(array) of a particular data type.

For example

```
int *arr = new int[10];
```

Delete operator

We delete the allocated space in C++ using the delete operator.

Syntax

```
delete pointer_variable_name
```

Example

```
delete m; // free m that is a variable
```

```
delete [] arr; // Release a block of memory
```

Example

```

#include <iostream>
using namespace std;

int main() {

    // declare an int pointer
    int* pointInt;

    // declare a float pointer
    float* pointFloat;

    // dynamically allocate memory
    pointInt = new int;
    pointFloat = new float;

    // assigning value to the memory
    *pointInt = 45;
    *pointFloat = 45.45f;

    cout << *pointInt << endl;
    cout << *pointFloat << endl;

    // deallocate the memory
    delete pointInt;
    delete pointFloat;

    return 0;
}

```

With arrays

// C++ Program to store GPA of n number of students and display it
// where n is the number of students entered by the user

```

#include <iostream>
using namespace std;

int main() {

    int num;
    cout << "Enter total number of students: ";
    cin >> num;
    float* ptr;

    // memory allocation of num number of floats
    ptr = new float[num];
}

```

```

cout << "Enter GPA of students." << endl;
for (int i = 0; i < num; ++i) {
    cout << "Student" << i + 1 << ": ";
    cin >> *(ptr + i);
}

cout << "\nDisplaying GPA of students." << endl;
for (int i = 0; i < num; ++i) {
    cout << "Student" << i + 1 << ": " << *(ptr + i) << endl;
}

// ptr memory is released
delete[] ptr;

return 0;
}

```

Class

We can think of class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object.

As, many houses can be made from the same description, we can create many objects from a class.

How to define a class in C++?

A class is defined in C++ using keyword class followed by the name of class.

The body of class is defined inside the curly brackets and terminated by a semicolon at the end.

```

class className
{
    // some data
    // some functions
};

```

Example: Class in C++

```

class Test
{
    private:
        int data1;
        float data2;

    public:
        void function1()

```



```

    { data1 = 2; }

float function2()
{
    data2 = 3.5;
    return data2;
}
};

```

Example: Class in C++

```

class Test
{
    private:
        int data1;
        float data2;

    public:
        void function1()
        { data1 = 2; }

        float function2()
        {
            data2 = 3.5;
            return data2;
        }
};

```

C++ Objects

To use the data and access functions defined in the class, you need to create objects.

Syntax to Define Object in C++

```
className objectVariableName;
```

Example

```

class Test
{
    private:
        int data1;
        float data2;

    public:
        void function1()
        { data1 = 2; }
}

```

```

float function2()
{
    data2 = 3.5;
    return data2;
}

};

int main()
{
    Test o1, o2;
}

```

How to access data member and member function in C++?

You can access the data members and member functions by using a . (dot) operator. For example,

```
o2.function1();
```

This will call the function1() function inside the Test class for objects o2.

Access Specifiers;

In C++, there are three access specifiers:

- public - members are accessible from outside the class
- private - members cannot be accessed (or viewed) from outside the class
- protected - members cannot be accessed from outside the class, however, they can be accessed in inherited classes.

1. Public Access Specifier

This keyword is used to declare the functions and variables public, and any part of the entire program can access it. The members and member methods declared public can be accessed by other classes and functions. The public members of a class can be accessed from anywhere in the program using the (.) with the object of that class.

2. Private Access Specifiers

The private keyword is used to create private variables or private functions. The private members can only be accessed from within the class. Only the member functions or the friend functions are allowed to access the private data of a class or the methods of a class.

Protected Access Specifiers

The protected keyword is used to create protected variables or protected functions. The protected members can be accessed within and from the derived/child class.

Friend Function

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

```
class className {
    ... ..
    friend returnType functionName(arguments);
    ... ..
}

#include <iostream>
using namespace std;

class Distance {
private:
    int meter;

    // friend function
    friend int addFive(Distance);

public:
    Distance() : meter(0) {}
};

// friend function definition
int addFive(Distance d) {

    //accessing private members from the friend function
    d.meter += 5;
    return d.meter;
}

int main() {
    Distance D;
    cout << "Distance: " << addFive(D);
    return 0;
}
```

Constructors

In C++, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C++ has the same name as class.

There can be three types of constructors in C++.

Default constructor

Parameterized constructor

Copy constructor

C++ Default Constructor

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

```
#include <iostream>
using namespace std;
class Employee
{
public:
    Employee()
    {
        cout<<"Default Constructor"<<endl;
    }
};
int main()
{
    Employee e1; //creating an object of Employee
    Employee e2;
    return 0;
}
```

C++ Parameterized Constructor

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

```
#include <iostream>
using namespace std;
class A {
private:
```

```

int num1, num2 ;
public:
A(int n1, int n2) {
    num1 = n1;
    num2 = n2;
}
void display() {
    cout<<"num1 = "<< num1 <<endl;
    cout<<"num2 = "<< num2 <<endl;
}
};
int main() {
A obj(3,8);
    obj.display();
    return 0;
}

```

Copy Constructor

In the C++ programming language, a copy constructor is a special constructor for creating a new object as a copy of an existing object. Copy constructors are the standard way of copying objects in C++,

Syntax of Copy Constructor

```

Classname (classname & objectname)

```

```

{
    . . . .
}

```

```

#include<iostream>

```

```

using namespace std;

```

```

class Samplecopyconstructor

```

```

{

```

```

private:
int x, y; //data members
public:
Samplecopyconstructor(int x1, int y1)
{
    x = x1;
    y = y1;
}
/* Copy constructor */
Samplecopyconstructor (Samplecopyconstructor &sam)
{
    x = sam.x;
    y = sam.y;
}
void display()
{
    cout<<x<<" "<<y<<endl;
}
};
/* main function */
int main()
{
    Samplecopyconstructor obj1(10, 15);    // Normal constructor
    Samplecopyconstructor obj2 = obj1;    // Copy constructor
    cout<<"Normal constructor : ";
    obj1.display();
    cout<<"Copy constructor : ";
    obj2.display();
    return 0;
}

```

Operator Overloading

Using **operator overloading** in C++, you can specify more than one meaning for an operator in one scope. The purpose of operator overloading is to provide a special meaning of an operator for a user-defined data type.

With the help of operator overloading, you can redefine the majority of the C++ operators. You can also use operator overloading to perform different operations using one operator.

Syntax

To overload a C++ operator, you should define a special function inside the Class as follows:

```
class class_name
{
    ... ..
    public
        return_type operator symbol (argument(s))
        {
            ... ..
        }
    ... ..
};
```

Can all C++ Operators be Overloaded?

No. There are C++ operators that can't be overloaded.

They include:

- :: -Scope resolution operator
- ?: -ternary operator.
- . -member selector
- sizeof operator
- * -member pointer selector

Here are rules for Operator Overloading:

- For it to work, at least one operand must be a user-defined class object.
- You can only overload existing operators. You can't overload new operators.
- Some operators cannot be overloaded using a friend function. However, such operators can be overloaded using member function.

```
// Overload ++ when used as prefix
```

```
#include <iostream>
```

```
using namespace std;

class Count {
    private:
        int value;
    public:
        // Constructor to initialize count to 5
        Count() : value(5) {}
        // Overload ++ when used as prefix
        void operator ++ () {
            value=value+100;
        }
        void display() {
            cout << "Count: " << value << endl;
        }
};

int main() {
    Count count1;
    // Call the "void operator ++ ()" function
    ++count1;
    count1.display();
    return 0;
}
```


BINARY OPERATOR OVERLOADING BY CONCATENATING 2 STRINGS

// C++ Program to concatenate two string using unary operator overloading

```
#include <iostream>
```

```
#include <string.h>
```

```
using namespace std;
```

```
// Class to implement operator overloading
```

```
// function for concatenating the strings
```

```
class AddString {
```

```
public:
```

```
    // Classes object of string
```

```
    char s1[25], s2[25];
```

```
    // Parameterized Constructor
```

```
    AddString(char str1[], char str2[])
```

```
    {
```

```
        // Initialize the string to class object
```

```
        strcpy(this->s1, str1);
```

```
        strcpy(this->s2, str2);
```

```
    }
```

```
    // Overload Operator+ to concat the string
```

```
    void operator+()
```

```
    {
```

```

        cout << "\nConcatenation: " << strcat(s1, s2);
    }
};

// Driver Code
int main()
{
    // Declaring two strings
    char str1[] = "Geeks";
    char str2[] = "ForGeeks";

    // Declaring and initializing the class
    // with above two strings
    AddString a1(str1, str2);

    // Call operator function
    +a1;

    return 0;
}

```

'**Class Template** can also be defined similarly to the Function Template. When a class uses the concept of Template, then the class is known as generic class.

'Syntax

```

template<class Ttype>

class class_name

{ }

```

'**Ttype** is a placeholder name which will be determined when the class is instantiated. We can define more than one generic data type using a comma-separated list. The Ttype can be used inside the class body.

'Now, we create an instance of a class

```

class_name<type> ob;

```

'**where class_name**: It is the name of the class.

'**type**: It is the type of the data that the class is operating on.

'**ob**: It is the name of the object.

```
#include <iostream>
using namespace std;
template<class T>
class A
{
    public:
    T num1 = 5;
    T num2 = 6;
    void add()
    {
        std::cout << "Addition of num1 and num2 : " << num1+num2<<std::endl;
    }
};

int main()
{
    A<int> d;
    d.add();
    return 0;
}
```

CLASS TEMPLATE WITH MULTIPLE PARAMETERS

'We can use more than one generic data type in a class template, and each generic data type is separated by the comma.

Syntax

```
template<class T1, class T2, .....>
```

```
class class_name
```

```
{
```

```
    // Body of the class.
```

```
}
```

```
#include <iostream>
```

```
using namespace std;
```

```

template<class T1, class T2>
class A
{
    T1 a;
    T2 b;
    public:
    A(T1 x,T2 y)
    {
        a = x;
        b = y;
    }
}

void display()
{
    std::cout << "Values of a and b are : " << a<<" , "<<b<<std::endl;
}

};

int main()
{
    A<int,float> d(5,6.5);
    d.display();
    return 0;
}

```

Nontype Template Arguments

The template can contain multiple arguments, and we can also use the non-type arguments. In addition to the type T argument, we can also use other types of arguments such as strings, function names, constant expression and built-in types.

```

template<class T, int size>

class array
{
    T arr[size];    // automatic array initialization.
};

```

In the above case, the nontype template argument is size and therefore, template supplies the size of the array as an argument.

Arguments are specified when the objects of a class are created:

```
'array<int, 15> t1;           // array of 15 integers.  
'array<float, 10> t2;        // array of 10 floats.  
'array<char, 4> t3;          // array of 4 chars.
```

```
#include <iostream>  
using namespace std;  
template<class T, int size>  
class A  
{  
    public:  
    T arr[size];  
    void insert()  
    {  
        int i=1;  
        for (int j=0;j<size;j++)  
        {  
            arr[j] = i;  
            i++;  
        }  
    }  
    void display()  
    {  
        for(int i=0;i<size;i++)  
        {  
            std::cout << arr[i] << " ";  
        }  
    }  
};  
int main()  
{  
    A<int,10> t1;  
    t1.insert();  
    t1.display();  
    return 0;  
}
```

