Introduction to C++ Classes and Objects

The classes are the most important feature of C++ that leads to Object Oriented Programming. Class is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating instance of that class.

The variables inside class definition are called as data members and the functions are called member functions.

For example: Class of birds, all birds can fly and they all have wings and beaks. So here flying is a behavior and wings and beaks are part of their characteristics. And there are many different birds in this class with different names but they all posses this behavior and characteristics.

Similarly, class is just a blue print, which declares and defines characteristics and behavior, namely data members and member functions respectively. And all objects of this class will share these characteristics and behavior.

More about Classes

1. Class name must start with an uppercase letter(Although this is not mandatory). If class name is made of more than one word, then first letter of each word must be in uppercase. *Example*,

   ```
   class Study, class StudyTonight etc
   ```

   Copy

2. Classes contain, data members and member functions, and the access of these data members and variable depends on the access specifiers (discussed in next section).
3. Class's member functions can be defined inside the class definition or outside the class definition.
4. Class in C++ are similar to structures in C, the only difference being, class defaults to private access control, where as structure defaults to public.
5. All the features of OOPS, revolve around classes in C++. Inheritance, Encapsulation, Abstraction etc.
6. Objects of class holds separate copies of data members. We can create as many objects of a class as we need.
7. Classes do posses more characteristics, like we can create abstract classes, immutable classes, all this we will study later.

Objects of Classes

Class is mere a blueprint or a template. No storage is assigned when we define a class. Objects are instances of class, which holds the data variables declared in class and the member functions work on these class objects.

Each object has different data variables. Objects are initialised using special class functions called **Constructors**. We will study about constructors later.

And whenever the object is out of its scope, another special class member function called **Destructor** is called, to release the memory reserved by the object. C++ doesn't have Automatic Garbage Collector like in JAVA, in C++ Destructor performs this task.

```
class Abc

{

    int x;

    void display()

    {

        // some statement

    }

};


int main()

{

    Abc obj;    // Object of class Abc created

}
```

Copy

Access Control in C++

Now before studying how to define class and its objects, lets first quickly learn what are access modifiers.

Access modifiers in C++ class defines the access control rules. C++ has 3 new keywords introduced, namely,

1. public
2. private
3. protected

These access modifiers are used to set boundaries for availability of members of class be it data members or member functions

Access modifiers in the program, are followed by a colon. You can use either one, two or all 3 modifiers in the same class to set different boundaries for different class members. They change the boundary for all the declarations that follow them.

Public Access Modifier in C++

Public, means all the class members declared under **public** will be available to everyone. The data members and member functions declared public can be accessed by other classes too. Hence there are chances that they might change them. So the key members must not be declared public.

```
class PublicAccess

{

    // public access modifier

    public:

    int x;              // Data Member Declaration

    void display();     // Member Function decaration

}
```

Copy

Private Access Modifier in C++

Private keyword, means that no one can access the class members declared **private**, outside that class. If someone tries to access the private members of a class, they will get a **compile time error**. By default class variables and member functions are private.

```
class PrivateAccess

{

    // private access modifier

    private:

    int x;              // Data Member Declaration

    void display();     // Member Function decaration

}
```

Copy

Protected Access Modifier in C++

Protected, is the last access specifier, and it is similar to private, it makes class member inaccessible outside the class. But they can be accessed by any subclass of that class. (If class A is **inherited** by class B, then class B is subclass of class A. We will learn about inheritance later.)

```
class ProtectedAccess

{

    // protected access modifier

    protected:
```

```
    int x;              // Data Member Declaration

    void display();     // Member Function decaration

}
```

Defining Class and Creating Objects

When we define any class, we are not defining any data, we just define a structure or a blueprint, as to what the object of that class type will contain and what operations can be performed on that object.

Below is the syntax of class definition,

```
class ClassName

{

    Access specifier:

    Data members;

    Member Functions()

    {

        // member function defintion

    }

};
```

Copy

Here is an example, we have made a simple class named Student with appropriate members,

```
class Student

{

    public:

    int rollno;

    string name;

};
```

Copy

So its clear from the syntax and example, class definition starts with the keyword "class" followed by the class name. Then inside the curly braces comes the class body, that is data members and member functions, whose access is bounded by access specifier. A class definition ends with a semicolon, or with a list of object declarations.

For example:

```
class Student
```

```
{

    public:

    int rollno;

    string name;

}A,B;
```

Copy

Here A and B are the objects of class Student, declared with the class definition. We can also declare objects separately, like we declare variable of primitive data types. In this case the data type is the class name, and variable is the object.

```
int main()

{

    // creating object of class Student

    Student A;

    Student B;

}
```

Copy

Both A and B will have their own copies of data members i.e. rollno and name and we can store different values for them in these objects.

Accessing Data Members of Class in C++

Accessing a data member depends solely on the access control of that data member. If its public, then the data member can be easily accessed using the direct member access (.) operator with the object of that class.

If, the data member is defined as private or protected, then we cannot access the data variables directly. Then we will have to create special public member functions to access, use or initialize the private and protected data members. These member functions are also called **Accessors** and **Mutator** methods or **getter** and **setter** functions.

Accessing Public Data Members

Following is an example to show you how to initialize and use the public data members using the dot (.) operator and the respective object of class.

```
class Student

{

    public:

    int rollno;
    string name;

};


int main()

{

    Student A;

    Student B;


    // setting values for A object

    A.rollno=1;

    A.name="Adam";


    // setting values for B object

    B.rollno=2;

    B.name="Bella";


    cout <<"Name and Roll no of A is: "<< A.name << "-" << A.rollno;

    cout <<"Name and Roll no of B is: "<< B.name << "-" << B.rollno;

}
```

Copy

```
Name and Roll no of A is: Adam-1

Name and Roll no of B is: Bella-2
```

Accessing Private Data Members

To access, use and initialize the private data member you need to create getter and setter functions, to get and set the value of the data member.

The setter function will set the value passed as argument to the private data member, and the getter function will return the value of the private data member to be used. Both getter and setter function must be defined public.

Example :

```
class Student
```

```
{
    private:    // private data member
    int rollno;

    public:
    // public function to get value of rollno - getter
    int getRollno()
    {
        return rollno;
    }
    // public function to set value for rollno - setter
    void setRollno(int i)
    {
        rollno=i;
    }
};

int main()
{
    Student A;
    A.rollono=1;  //Compile time error
    cout<< A.rollno; //Compile time error

    A.setRollno(1);  //Rollno initialized to 1
    cout<< A.getRollno(); //Output will be 1
}
```
Copy

So this is how we access and use the private data members of any class using the getter and setter methods. We will discuss this in more details later.

Accessing Protected Data Members

Protected data members, can be accessed directly using dot (.) operator inside the **subclass** of the current class, for non-subclass we will have to follow the steps same as to access private data member.

Member Functions of Classes in C++

Member functions are the functions, which have their declaration inside the class definition and works on the data members of the class. The definition of member functions can be inside or outside the definition of class.

If the member function is defined inside the class definition it can be defined directly, but if its defined outside the class, then we have to use the scope resolution :: operator along with class name alng with function name.

For example:

```
class Cube

{

    public:

    int side;

    /*

        Declaring function getVolume

        with no argument and return type int.

    */

    int getVolume();

};
```

Copy

If we define the function inside class then we don't not need to declare it first, we can directly define the function.

```
class Cube

{

    public:

    int side;

    int getVolume()

    {

        return side*side*side;      //returns volume of cube

    }

};
```

Copy

But if we plan to define the member function outside the class definition then we must declare the function inside class definition and then define it outside.

```
class Cube

{

    public:

    int side;

    int getVolume();

}


// member function defined outside class definition

int Cube :: getVolume()

{

    return side*side*side;

}
```

Copy

The main function for both the function definition will be same. Inside main() we will create object of class, and will call the member function using dot . operator.

Calling Class Member Function in C++

Similar to accessing a data member in the class, we can also access the public member functions through the class object using the dot operator (.).

Below we have a simple code example, where we are creating an object of the class `Cube` and calling the member function `getVolume()`:

```
int main()

{

    Cube C1;

    C1.side = 4;     // setting side value

    cout<< "Volume of cube C1 = "<< C1.getVolume();

}
```

Copy

```
Volume of cube C1 = 16
```

Similarly we can define the getter and setter functions to access private data members, inside or outside the class definition.

Types of Class Member Functions in C++

We already know what member functions are, what they do, how to define member functions and how to call them using class objects. Now lets learn about some special member functions which can be defined in C++ classes. Following are the different types of Member functions:

1. Simple functions
2. Static functions
3. Const functions
4. Inline functions
5. Friend functions

Simple Member functions in C++

These are the basic member function, which dont have any special keyword like static etc as prefix. All the general member functions, which are of below given form, are termed as simple and basic member functions.

```
return_type functionName(parameter_list)

{

    function body;

}
```

Copy

Static Member functions in C++

Static is something that holds its position. Static is a keyword which can be used with data members as well as the member functions. We will discuss this in details later. As of now we will discuss its usage with member functions only.

A function is made static by using `static` keyword with function name. These functions work for the class as whole rather than for a particular object of a class.

It can be called using the object and the direct member access `.` operator. But, its more typical to call a static member function by itself, using class name and scope resolution `::` operator.

For example:

```
class X

{

    public:
```

```
    static void f()
    {
        // statement
    }
};

int main()
{
    X::f();   // calling member function directly with class name
}
```

Copy

These functions cannot access ordinary data members and member functions, but only `static` data members and `static` member functions can be called inside them.

It doesn't have any "this" keyword which is the reason it cannot access ordinary members. We will study about "this" keyword later.

Const Member functions in C++

We will study **Const** keyword in detail later(Const Keyword), but as an introduction, Const keyword makes variables constant, that means once defined, there values can't be changed.

When used with member function, such member functions can never modify the object or its related data members.

```
// basic syntax of const Member Function

void fun() const
{
    // statement
}
```

Copy

Inline functions in C++

All the member functions defined inside the class definition are by default declared as Inline. We will study Inline Functions in details in the next topic.

Friend functions in C++

Friend functions are actually not class member function. Friend functions are made to give **private** access to non-class functions. You can declare a global function as friend, or a member function of other class as friend.

For example:

```
class WithFriend

{

    int i;

    public:

    friend void fun(); // global function as friend

};


void fun()

{

    WithFriend wf;

    wf.i=10;  // access to private data member

    cout << wf.i;

}


int main()

{

    fun(); //Can be called directly

}
```

Copy
Hence, friend functions can access private data members by creating object of the class. Similarly we can also make function of some other class as friend, or we can also make an entire class as **friend class**.

```
class Other

{

    void fun();

};
```

```
class WithFriend

{

    private:

    int i;

    public:

    void getdata();  // Member function of class WithFriend


    // making function of class Other as friend here

    friend void Other::fun();


    // making the complete class as friend

    friend class Other;

};
```

Copy

When we make a class as friend, all its member functions automatically become friend functions.

Friend Functions is a reason, why C++ is not called as a **pure Object Oriented language**. Because it violates the concept of **Encapsulation**.

Inline Functions in C++

All the member functions defined inside the class definition are by default declared as Inline. Let us have some background knowledge about these functions.

You must remember Preprocessors from C language. Inline functions in C++ do the same thing what Macros did in C language. Preprocessors/Macros were not used in C++ because they had some drawbacks.

Drawbacks of Preprocessors/Macros in C++

In Macro, we define certain variable with its value at the beginning of the program, and everywhere inside the program where we use that variable, its replaced by its value on Compilation.

1) Problem with spacing

Let us try to understand this problem using an example,

```
#define G (y) (y+1)
```

Copy

Here we have defined a Macro with name `G(y)`, which is to be replaced by its value, that is `(y+1)` during compilation. But, what actually happens when we call `G(y)`,

```
G(1)  // Macro will replace it
```
Copy
the preprocessor will expand it like,

```
(y) (y+1) (1)
```
Copy
You must be thinking why this happened, this happened because of the spacing in Macro definition. Hence big functions with several expressions can never be used with macro, so Inline functions were introduced in C++.

2) Complex Argument Problem

In some cases such Macro expressions work fine for certain arguments but when we use complex arguments problems start arising.

```
#define MAX(x,y) x>y?1:0
```
Copy
Now if we use the expression,

```
if(MAX(a&0x0f, 0x0f))  // Complex Argument
```
Copy
Macro will Expand to,

```
if( a&0x0f > 0x0f ? 1:0)
```
Copy
Here precedence of operators will lead to problem, because precedence of `&` is lower than that of `>`, so the macro evaluation will surprise you. This problem can be solved though using parenthesis, but still for bigger expressions problems will arise.

3) No way to access Private Members of Class

With Macros, in C++ you can never access private variables, so you will have to make those members public, which will expose the implementation.

```
class Y

{

    int x;

    public :

    #define VAL(Y::x)   // Its an Error

}
```
Copy

Inline Functions in C++

Inline functions are actual functions, which are copied everywhere during compilation, like preprocessor macro, so the overhead of function calling is reduced. All the functions defined inside class definition are by default inline, but you can also make any non-class function inline by using keyword **inline** with them.

For an inline function, declaration and definition must be done together. For example,

```
inline void fun(int a)

{

    return a++;

}
```

Copy

Some Important points about Inline Functions

1. We must keep inline functions small, small inline functions have better efficiency.
2. Inline functions do increase efficiency, but we should not make all the functions inline. Because if we make large functions inline, it may lead to **code bloat**, and might affect the speed too.
3. Hence, it is adviced to define large functions outside the class definition using scope resolution `::` operator, because if we define such functions inside class definition, then they become inline automatically.
4. Inline functions are kept in the Symbol Table by the compiler, and all the call for such functions is taken care at compile time.

Getter and Setter Functions in C++

We have already studied this in the topic accessing **private** data variables inside a class. We use access functions, which are inline to do so.

```
class Auto

{

    // by default private

    int price;



    public:

    // getter function for variable price

    int getPrice()

    {

        return price;
    }

    // setter function for variable price

    void setPrice(int x)

    {

        i=x;

    }

};
```

Copy

Here getPrice() and setPrice() are inline functions, and are made to access the private data members of the class Auto. The function getPrice(), in this case is called **Getter or Accessor** function and the function setPrice() is a **Setter or Mutator** function.

There can be overlaoded Accessor and Mutator functions too. We will study overloading functions in next topic.

Limitations of Inline Functions

1. Large Inline functions cause Cache misses and affect performance negatively.
2. Compilation overhead of copying the function body everywhere in the code on compilation, which is negligible for small programs, but it makes a difference in large code bases.
3. Also, if we require address of the function in program, compiler cannot perform inlining on such functions. Because for providing address to a function, compiler will have to allocate storage to it. But inline functions doesn't get storage, they are kept in Symbol table.

Understanding Forward References in C++

All the inline functions are evaluated by the compiler, at the end of class declaration.

```
class ForwardReference

{

    int i;

    public:

    // call to undeclared function

    int f()

    {

        return g()+10;

    }
    int g()

    {

        return i;

    }

};



int main()

{

    ForwardReference fr;

    fr.f();

}
```

Copy

You must be thinking that this will lead to compile time error, but in this case it will work, because no inline function in a class is evaluated until the closing braces of class declaration.

Function Overloading in C++

If any class have multiple functions with same names but different parameters then they are said to be overloaded. Function overloading allows you to use the same name for different functions, to perform, either same or different functions in the same class.

Function overloading is usually used to enhance the readability of the program. If you have to perform one single operation but with different number or types of arguments, then you can simply overload the function.

Different ways to Overload a Function

1. By changing number of Arguments.
2. By having different types of argument.

Function Overloading: Different Number of Arguments

In this type of function overloading we define two functions with same names but different number of parameters of the same type. For example, in the below mentioned program we have made two sum() functions to return sum of two and three integers.

```
// first definition

int sum (int x, int y)

{
    cout << x+y;

}



// second overloaded defintion

int sum(int x, int y, int z)

{

    cout << x+y+z;

}
```

Copy

Here sum() function is said to overloaded, as it has two defintion, one which accepts two arguments and another which accepts three arguments. Which sum() function will be called, depends on the number of arguments.

```
int main()

{

    // sum() with 2 parameter will be called

    sum (10, 20);



    //sum() with 3 parameter will be called

    sum(10, 20, 30);

}
```

Copy

```

30

60
```

Function Overloading: Different Datatype of Arguments

In this type of overloading we define two or more functions with same name and same number of parameters, but the type of parameter is different. For example in this program, we have two sum() function, first one gets two integer arguments and second one gets two double arguments.

```
// first definition

int sum(int x, int y)

{

    cout<< x+y;
}


// second overloaded defintion

double sum(double x, double y)

{

    cout << x+y;

}


int main()

{

    sum (10,20);

    sum(10.5,20.5);

}
```
Copy

```

30

31.0
```

Functions with Default Arguments

When we mention a default value for a parameter while declaring the function, it is said to be as default argument. In this case, even if we make a call to the function without passing any value for that parameter, the function will take the default value specified.

```
sum(int x, int y=0)

{

    cout << x+y;

}
```
Copy
Here we have provided a default value for y, during function definition.

```


int main()

{

    sum(10);
    sum(10,0);

    sum(10,10);

}
```
Copy

```
10 10 20
```

First two function calls will produce the exact same value.

for the third function call, y will take 10 as value and output will become 20.

By setting default argument, we are also overloading the function. Default arguments also allow you to use the same function in different situations just like function overloading.

Rules for using Default Arguments

1. Only the last argument must be given default value. You cannot have a default argument followed by non-default argument.

```
2.

3. sum (int x,int y);

4. sum (int x,int y=0);

5. sum (int x=0,int y);  // This is Incorrect
```

Copy

6. If you default an argument, then you will have to default all the subsequent arguments after that.

```
7.

8. sum (int x,int y=0);

9. sum (int x,int y=0,int z);  // This is incorrect

10.sum (int x,int y=10,int z=10);  // Correct
```

Copy

11. You can give any value a default value to argument, compatible with its datatype.

Function with Placeholder Arguments

When arguments in a function are declared without any identifier they are called placeholder arguments.

```
void sum (int, int);
```
Copy

Such arguments can also be used with default arguments.

```
void sum (int, int=0);
```

Constructors and Destructors in C++

Constructors are special class functions which performs initialization of every object. The Compiler calls the Constructor whenever an object is created. Constructors initialize values to object members after storage is allocated to the object.

Whereas, Destructor on the other hand is used to destroy the class object.

Before moving forward with Constructors and Destructors in C++ language, check these topics out to understand the concept better:

- Function in C++
- Class and Objects in C++
- Data Members

Let's start with Constructors first, following is the syntax of defining a constructor function in a class:

```
class A

{

    public:

    int x;

    // constructor

    A()

    {

        // object initialization

    }

};
```

Copy
While defining a contructor you must remeber that the **name of constructor** will be same as the **name of the class**, and contructors will never have a return type.

Constructors can be defined either inside the class definition or outside class definition using class name and scope resolution :: operator.

```
class A

{

    public:

    int i;

    A(); // constructor declared

};
```

```
// constructor definition
A::A()
{
    i = 1;
}
```
Copy

Types of Constructors in C++

Constructors are of three types:

1. Default Constructor
2. Parametrized Constructor
3. Copy COnstructor

Default Constructors

Default constructor is the constructor which doesn't take any argument. It has no parameter.

Syntax:

```
class_name(parameter1, parameter2, ...)
{
    // constructor Definition
}
```
Copy
For example:

```
class Cube
{
    public:
    int side;
    Cube()
    {
        side = 10;
    }
};


int main()

{

    Cube c;

    cout << c.side;

}
```

Copy

```
10
```

In this case, as soon as the object is created the constructor is called which initializes its data members.

A default constructor is so important for initialization of object members, that even if we do not define a constructor explicitly, the compiler will provide a default constructor implicitly.

```
class Cube

{

    public:

    int side;

};


int main()

{

    Cube c;

    cout << c.side;

}
```

Copy

```
0 or any random value
```

In this case, default constructor provided by the compiler will be called which will initialize the object data members to default value, that will be 0 or any random integer value in this case.

Parameterized Constructors

These are the constructors with parameter. Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.

For example:

```
class Cube
{
    public:
    int side;
    Cube(int x)
    {
        side=x;
    }
};

int main()
{
    Cube c1(10);
    Cube c2(20);
    Cube c3(30);
    cout << c1.side;
    cout << c2.side;
    cout << c3.side;
}
```

Copy

```
10
20
30
```

By using parameterized construcor in above case, we have initialized 3 objects with user defined values. We can have any number of parameters in a constructor.

Copy Constructors

These are special type of Constructors which takes an object as argument, and is used to copy values of data members of one object into other object. We will study copy constructors in detail later.

Constructor Overloading in C++

Just like other member functions, constructors can also be overloaded. Infact when you have both default and parameterized constructors defined in your class you are having Overloaded Constructors, one with no parameter and other with parameter.

You can have any number of Constructors in a class that differ in parameter list.

```
class Student

{

    public:

    int rollno;

    string name;

    // first constructor

    Student(int x)

    {

        rollno = x;

        name = "None";

    }

    // second constructor

    Student(int x, string str)

    {

        rollno = x;

        name = str;

    }

};


int main()

{

    // student A initialized with roll no 10 and name None

    Student A(10);


    // student B initialized with roll no 11 and name John

    Student B(11, "John");
}
```

Copy
In above case we have defined two constructors with different parameters, hence overloading the constructors.

One more important thing, if you define any constructor explicitly, then the compiler will not provide default constructor and you will have to define it yourself.

In the above case if we write `Student S;` in **main()**, it will lead to a compile time error, because we haven't defined default constructor, and compiler will not provide its default constructor because we have defined other parameterized constructors.

Destructors in C++

Destructor is a special class function which destroys the object as soon as the scope of object ends. The destructor is called automatically by the compiler when the object goes out of scope.

The syntax for destructor is same as that for the constructor, the class name is used for the name of destructor, with a **tilde** `~` sign as prefix to it.

```
class A

{

    public:

    // defining destructor for class

    ~A()

    {

        // statement

    }

};
```

Copy
Destructors will never have any arguments.

Example to see how Constructor and Destructor are called

Below we have a simple class `A` with a constructor and destructor. We will create object of the class and see when a constructor is called and when a destructor gets called.

```
class A

{

    // constructor

    A()
    {
        cout << "Constructor called";
    }

    // destructor
    ~A()
    {
        cout << "Destructor called";
    }
};

int main()
{
    A obj1;   // Constructor Called
    int x = 1
    if(x)
    {
        A obj2;  // Constructor Called
    }   // Destructor Called for obj2
} //  Destructor called for obj1
```

Copy

```
Constructor called
Constructor called
Destructor called
Destructor called
```

When an object is created the constructor of that class is called. The object reference is destroyed when its scope ends, which is generally after the closing curly bracket `}` for the code block in which it is created.

The object `obj2` is destroyed when the `if` block ends because it was created inside the `if` block. And the object `obj1` is destroyed when the `main()` function ends.

Single Definition for both Default and Parameterized Constructor

In this example we will use **default argument** to have a single definition for both defualt and parameterized constructor.

```
class Dual

{

    public:

    int a;

    Dual(int x=0)

    {

        a = x;

    }

};


int main()

{

    Dual obj1;

    Dual obj2(10);

}
```

Copy

Here, in this program, a single Constructor definition will take care for both these object initializations. We don't need separate default and parameterized constructors.

Static Keyword in C++

Static is a keyword in C++ used to give special characteristics to an element. Static elements are allocated storage only once in a program lifetime in static storage area. And they have a scope till the program lifetime. Static Keyword can be used with following,

1. Static variable in functions
2. Static Class Objects
3. Static member Variable in class
4. Static Methods in class

Static Variables inside Functions

Static variables when used inside function are initialized only once, and then they hold there value even through function calls.

These static variables are stored on static storage area , not in stack.

```
void counter()

{

    static int count=0;

    cout << count++;

}



int main(0

{

    for(int i=0;i<5;i++)

    {

        counter();

    }

}
```
Copy

```
0 1 2 3 4
```
Let's se the same program's output **without using static** variable.

```
void counter()

{

    int count=0;

    cout << count++;

}



int main(0

{

    for(int i=0;i<5;i++)

    {

        counter();

    }

}
```
Copy

```
0 0 0 0 0
```

If we do not use static keyword, the variable count, is reinitialized everytime when counter() function is called, and gets destroyed each time when counter() functions ends. But, if we make it static, once initialized count will have a scope till the end of main() function and it will carry its value through function calls too.

If you don't initialize a static variable, they are by default initialized to zero.

Static Class Objects

Static keyword works in the same way for class objects too. Objects declared static are allocated storage in static storage area, and have scope till the end of program.

Static objects are also initialized using constructors like other normal objects. Assignment to zero, on using static keyword is only for primitive datatypes, not for user defined datatypes.

```
class Abc

{

    int i;

    public:

    Abc()

    {

        i=0;

        cout << "constructor";

    }

    ~Abc()

    {

        cout << "destructor";

    }

};



void f()

{

    static Abc obj;

}
```

```
int main()

{

    int x=0;

    if(x==0)

    {

        f();

    }

    cout << "END";

}
```

Copy

```
constructor END destructor
```

You must be thinking, why was the destructor not called upon the end of the scope of `if` condition, where the reference of object `obj` should get destroyed. This is because object was `static`, which has scope till the program's lifetime, hence destructor for this object was called when `main()` function exits.

Static Data Member in Class

Static data members of class are those members which are shared by all the objects. Static data member has a single piece of storage, and is not available as separate copy with each object, like other non-static data members.

Static member variables (data members) are not initialied using constructor, because these are not dependent on object initialization.

Also, it must be initialized explicitly, always outside the class. If not initialized, Linker will give error.

```
class X

{

    public:

    static int i;

    X()

    {

        // construtor

    };

};
```

```
int X::i=1;


int main()
{
    X obj;
    cout << obj.i;   // prints value of i
}
```

Copy

```
1
```

Once the definition for static data member is made, user cannot redefine it. Though, arithmetic operations can be performed on it.

Static Member Functions

These functions work for the class as whole rather than for a particular object of a class.

It can be called using an object and the direct member access . operator. But, its more typical to call a static member function by itself, using class name and scope resolution :: operator.

For example:

```
class X
{
    public:
    static void f()
    {
        // statement
    }
};


int main()
{
    X::f();   // calling member function directly with class name
}
```

Copy
These functions cannot access ordinary data members and member functions, but only static data members and static member functions.

It doesn't have any "this" keyword which is the reason it cannot access ordinary members. We will study about "this" keyword later.

const Keyword in C++

Constant is something that doesn't change. In C language and C++ we use the keyword const to make program elements constant. const keyword can be used in many contexts in a C++ program. It can be used with:

1. Variables
2. Pointers
3. Function arguments and return types
4. Class Data members
5. Class Member functions
6. Objects

---

1) Constant Variables in C++

If you make any variable as constant, using const keyword, you cannot change its value. Also, the constant variables must be initialized while they are declared.

```
int main

{

    const int i = 10;

    const int j = i + 10;      // works fine

    i++;    // this leads to Compile time error

}
```

Copy
In the above code we have made i as constant, hence if we try to change its value, we will get compile time error. Though we can use it for substitution for other variables.

---

2) Pointers with const keyword in C++

Pointers can be declared using **const** keyword too. When we use const with pointers, we can do it in two ways, either we can apply const to what the pointer is pointing to, or we can make the pointer itself a constant.

Pointer to a const variable

This means that the pointer is pointing to a const variable.

```
const int* u;
```

Copy

Here, u is a pointer that can point to a const int type variable. We can also write it like,

```
char const* v;
```

Copy

still it has the same meaning. In this case also, v is a pointer to an char which is of const type.

Pointers to a const variable is very useful, as this can be used to make any string or array immutable(i.e they cannot be changed).

const Pointer

To make a pointer constant, we have to put the const keyword to the right of the *.

```
int x = 1;

int* const w = &x;
```

Copy

Here, w is a pointer, which is const, that points to an int. Now we can't change the pointer, which means it will always point to the variable x but can change the value that it points to, by changing the value of x.

The constant pointer to a variable is useful where you want a storage that can be changed in value but not moved in memory. Because the pointer will always point to the same memory location, because it is defined with const keyword, but the value at that memory location can be changed.

NOTE: We can also have a const pointer pointing to a const variable.

```
const int* const x;
```

---

3) const Function Arguments and Return types

We can make the return type or arguments of a function as const. Then we cannot change any of them.

```
void f(const int i)

{

    i++;    // error

}



const int g()

{

    return 1;
}
```

Copy

Some Important points to Remember

1. For built in datatypes, returning a const or non-const value, doesn't make any difference.

```
const int h()
{
    return 1;
}

int main()
{
    const int j = h();
    int k = h();
```

```
}
```

Copy

Both j and k will be assigned the value 1. No error will occur.

11. For user defined datatypes, returning const, will prevent its modification.
12. Temporary objects created while program execution are always of const type.
13. If a function has a non-const parameter, it cannot be passed a const argument while making a call.

```
void t(int*)
{
    // function logic
```

```
}
```

Copy

If we pass a const int* argument to the function t, it will give error.

17. But, a function which has a const type parameter, can be passed a const type argument as well as a non-const argument.

```
void g(const int*)
{
    // function logic
}
```
Copy

This function can have a int* as well as const int* type argument.

---

4) Defining Class Data members as const

These are data variables in class which are defined using const keyword. They are not initialized during declaration. Their initialization is done in the constructor.

```
class Test

{

    const int i;

    public:

    Test(int x):i(x)

    {

        cout << "\ni value set: " << i;

    }

};


int main()

{

    Test t(10);

    Test s(20);

}
```
Copy

In this program, i is a constant data member, in every object its independent copy will be present, hence it is initialized with each object using the constructor. And once initialized, its value cannot be changed. The above way of initializing a class member is known as **Initializer List in C++**.

---

5) Defining Class Object as const

When an object is declared or created using the const keyword, its data members can never be changed, during the object's lifetime.

Syntax:

```
const class_name object;
```

Copy
For example, if in the class Test defined above, we want to define a constant object, we can do it like:

```
const Test r(30);
```

Copy

---

6) Defining Class's Member function as const

A const member functions never modifies data members in an object.

Syntax:

```
return_type function_name() const;
```

Copy

Example for const Object and const Member function

```
class StarWars

{

    public:

    int i;

    StarWars(int x)     // constructor

    {

        i = x;

    }


    int falcon() const  // constant function

    {

        /*

            can do anything but will not

            modify any data members

        */

        cout << "Falcon has left the Base";

    }


    int gamma()

    {

        i++;

    }

};


int main()

{

    StarWars objOne(10);        // non const object

    const StarWars objTwo(20);      // const object


    objOne.falcon();     // No error

    objTwo.falcon();     // No error


    cout << objOne.i << objTwo.i;


    objOne.gamma();     // No error

    objTwo.gamma();     // Compile time error

}
```

Copy

```
Falcon has left the Base

Falcon has left the Base

10 20
```

Here, we can see, that const member function never changes data members of class, and it can be used with both const and non-const objecta. But a const object cannot be used with a member function which tries to change its data members.

---

mutable Keyword

mutable keyword is used with member variables of class, which we want to change even if the object is of const type. Hence, mutable data members of a const objects can be modified.

```
class Zee

{
    int i;

    mutable int j;

    public:

    Zee()

    {

        i = 0;

        j = 0;

    }


    void fool() const

    {

        i++;    // will give error

        j++;    // works, because j is mutable

    }

};


int main()

{

    const Zee obj;

    obj.fool();

}
```

Copy

References in C++

References are like constant pointers that are automatically dereferenced. It is a new name given to an existing storage. So when you are accessing the reference, you are actually accessing that storage.

```
int main()

{

    int y=10;

    int &r = y;  // r is a reference to int y

    cout << r;
}
```

Copy

```
10
```

There is no need to use the * to dereference a reference variable.

Difference between Reference and Pointer

References	Pointers
Reference must be initialized when it is created.	Pointers can be initialized any time.
Once initialized, we cannot reinitialize a reference.	Pointers can be reinitialized any number of time.
You can never have a NULL reference.	Pointers can be NULL.
Reference is automatically dereferenced.	* is used to dereference a pointer.

References in Funtions

References are generally used for function argument lists and function return values, just like pointers.

Rules for using Reference in Functions

1. When we use reference in argument list, we must keep in mind that any change to the reference inside the function will cause change to the original argument outside th function.
2. When we return a reference from a function, you should see that whatever the reference is connected to shouldn't go out of scope when fnction ends. Either make that **global** or **static**

Example to explain the use of References

Below we have a simple code example to explain the use of references in C++,

```
int* first (int* x)
```

```
{

    (*x++);

    return x;   // SAFE, x is outside this scope

}


int& second (int& x)

{

    x++;

    return x;   // SAFE, x is outside this scope

}


int& third ()

{

    int q;

    return q;   // ERROR, scope of q ends here

}


int& fourth ()

{

    static int x;

    return x;   // SAFE, x is static, hence lives till the end.

}


int main()

{

    int a=0;

    first(&a);   // UGLY and explicit

    second(a);   // CLEAN and hidden

}
```
Copy
We have four different functions in the above program.

- **first()** takes a pointer as argument and returns a pointer, it will work fine. The returning pointer points to variable declared outside first(), hence it will be valid even after the first() ends.
- Similarly, **second()** will also work fine. The returning reference is connected to valid storage, that is `int a` in this case.
- But in case of **third()**, we declare a variable `q` inside the function and try to return a reference connected to it. But as soon as function third() ends, the local variable q is destroyed, hence nothing is returned.
- To remedify above problem, we make x as **static** in function **fourth()**, giving it a lifetime till main() ends, hence now a reference connected to x will be valid when returned.

Const Reference in C++

Const reference is used in function arguments to prevent the function from changing the argument.

```
void g(const int& x)

{

    x++;

}   // ERROR


int main()

{

    int i=10;

    g(i);

}
```
Copy
We cannot change the argument in the function because it is passed as const reference.

Argument Passing Guidelines

Usually **call by value** is used during funtcion call just to save our object or variable from being changed or modified, but whenever we pass an argument by value, its new copy is created. If we pass an object as argument then a copy of that object is created (constructor and destructors called), which affects efficiency.

Hence, we must use **const reference** type arguments. When we use, const reference, only an address is passed on stack, which is used inside the function and the function cannot change our argument because it is of const type.

So using const reference type argument reduces overhead and also saves our argument from being changed.

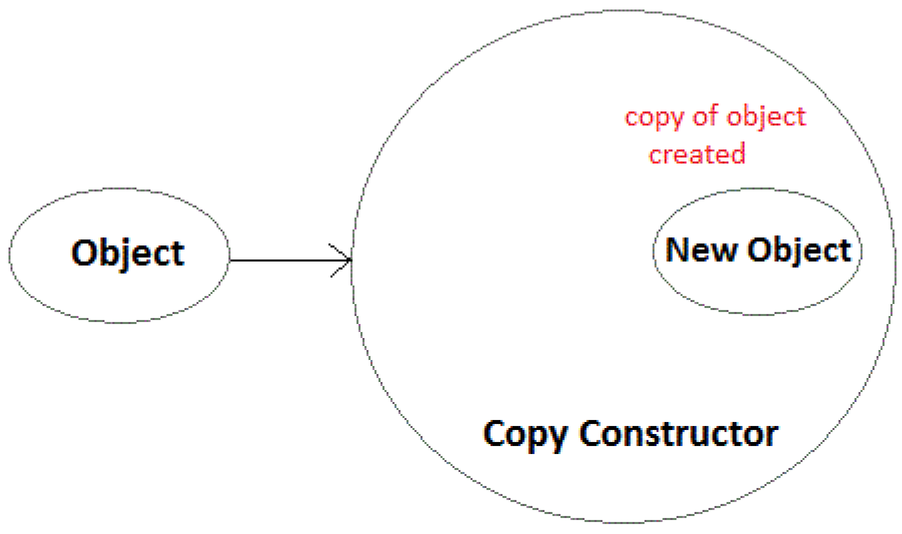
Copy Constructor in C++

Copy Constructors is a type of constructor which is used to create a copy of an already existing object of a class type. It is usually of the form **X (X&)**, where X is the class name. The compiler provides a default Copy Constructor to all the classes.

Syntax of Copy Constructor

```
Classname(const classname & objectname)

{

    . . . .

}
```

Copy
As it is used to create an object, hence it is called a constructor. And, it creates a new object, which is exact copy of the existing copy, hence it is called **copy constructor**.



Below is a sample program on Copy Constructor:

```
#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 (const 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;

}
```

Copy

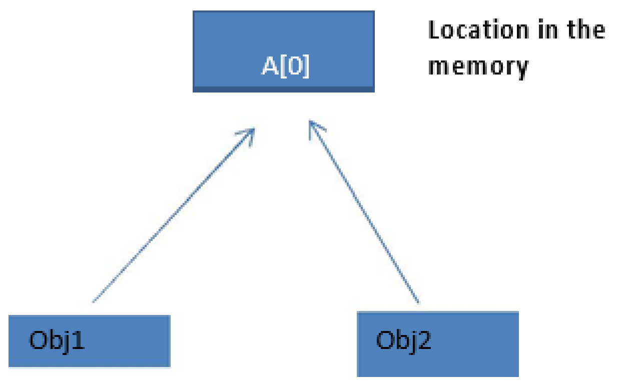
```
Normal constructor : 10 15

Copy constructor : 10 15
```

Shallow Copy Constructor

The concept of shallow copy constructor is explained through an example. Two students are entering their details in excel sheet simultaneously from two different machines shared over a network. Changes made by both of them will be reflected in the excel sheet. Because same excel sheet is opened in both locations. This is what happens in shallow copy constructor. Both objects will point to same memory location.

Shallow copy copies references to original objects. The compiler provides a default copy constructor. Default copy constructor provides a shallow copy as shown in below example. It is a bit-wise copy of an object.

Shallow copy constructor is used when class is not dealing with any dynamically allocated memory.



In the below example you can see both objects, c1 and c2, points to same memory location. When c1.concatenate() function is called, it affects c2 also. So both c1.display() and c2.display() will give same output.

```
#include<iostream>

#include<cstring>

using namespace std;

class CopyConstructor

{

    char *s_copy;

    public:

    CopyConstructor(const char *str)

    {

        s_copy = new char[16]; //Dynamic memory allocation

        strcpy(s_copy, str);

    }

    /* concatenate method */

    void concatenate(const char *str)

    {

        strcat(s_copy, str); //Concatenating two strings
    }
    /* copy constructor */
    ~CopyConstructor ()
    {
        delete [] s_copy;
    }
    void display()
    {
        cout<<s_copy<<endl;
    }
};
/* main function */
int main()
{
    CopyConstructor c1("Copy");
    CopyConstructor c2 = c1; //Copy constructor
    c1.display();
    c2.display();
    c1.concatenate("Constructor");    //c1 is invoking concatenate()
    c1.display();
    c2.display();
    return 0;
}
```

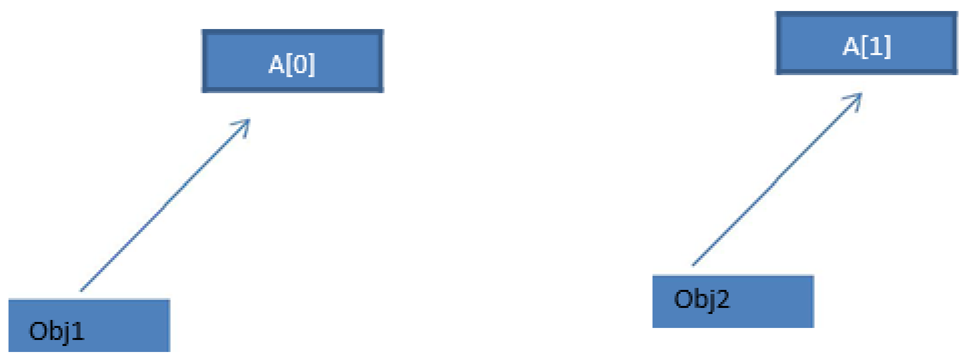
Copy

```
Copy
Copy
CopyConstructor
CopyConstructor
```

Deep Copy Constructor

Let's consider an example for explaining deep copy constructor. You are supposed to submit an assignment tomorrow and you are running short of time, so you copied it from your friend. Now you and your friend have same assignment content, but separate copies. Therefore any modifications made in your copy of assignment will not be reflected in your friend's copy. This is what happens in deep copy constructor.

Deep copy allocates separate memory for copied information. So the source and copy are different. Any changes made in one memory location will not affect copy in the other location. When we allocate dynamic memory using pointers we need user defined copy constructor. Both objects will point to different memory locations.



General requirements for deep copy:

- A normal constructor.
- A destructor to delete the dynamically allocated memory.
- A copy constructor to make a copy of the dynamically allocated memory.
- An overloaded assignment operator.

In the previous example you can see when c1 called concatenate(), changes happens in both c1 and c2, because both are pointing to same memory location.

In the below example you can see user defined copy constructor i.e deep copy constructor. Here both c1 and c2 points to different memory location. So changes made in one location will not affect the other.

```
#include<iostream>

#include<cstring>

using namespace std;

class CopyConstructor

{

    char *s_copy;

    public:

    CopyConstructor (const char *str)

    {

        s_copy = new char[16];   //Dynamic memory alocation

        strcpy(s_copy, str);

    }
    CopyConstructor (const CopyConstructor &str)

    {

        s_copy = new char[16]; //Dynamic memory alocation

        strcpy(s_copy, str.s_copy);

    }


    void concatenate(const char *str)

    {

        strcat(s_copy, str); //Concatenating two strings

    }


    ~CopyConstructor()

    {

        delete [] s_copy;

    }


    void display()

    {

        cout<<s_copy<<endl;

    }

};

/* main function */

int main()

{

    CopyConstructor c1("Copy");

    CopyConstructor c2 = c1;    //copy constructor

    c1.display();

    c2.display();

    c1.concatenate("Constructor");    //c1 is invoking concatenate()

    c1.display();
    c2.display();

    return 0;

}
```

Copy

```
Copy

Copy

CopyConstructor

Copy
```

Pointers to Class Members in C++

Just like pointers to normal variables and functions, we can have pointers to class member functions and member variables.

Let's see how this works.

Defining a Pointer of Class type

We can define pointer of class type, which can be used to point to class objects.

```
class Simple

{

    public:

    int a;

};


int main()

{

    Simple obj;

    Simple* ptr;   // Pointer of class type

    ptr = &obj;


    cout << obj.a;

    cout << ptr->a;  // Accessing member with pointer

}
```

Copy

Here you can see that we have declared a pointer of class type which points to class's object. We can access data members and member functions using pointer name with arrow `->` symbol.

Pointer to Data Members of Class

We can use pointer to point to class's data members (Member variables).

Syntax for Declaration :

```
datatype class_name :: *pointer_name;
```

Copy

Syntax for Assignment:

```
pointer_name = &class_name :: datamember_name;
```

Copy

Both declaration and assignment can be done in a single statement too.

```
datatype class_name::*pointer_name = &class_name::datamember_name ;
```

Copy

Using Pointers with Objects

For accessing normal data members we use the dot . operator with object and -> qith pointer to object. But when we have a pointer to data member, we have to dereference that pointer to get what its pointing to, hence it becomes,

```
Object.*pointerToMember
```

Copy

and with pointer to object, it can be accessed by writing,

```
ObjectPointer->*pointerToMember
```

Copy

Lets take an example, to understand the complete concept.

```
class Data

{

    public:

    int a;

    void print()

    {

        cout << "a is "<< a;

    }

};
```

```
int main()

{

    Data d, *dp;

    dp = &d;     // pointer to object


    int Data::*ptr=&Data::a;   // pointer to data member 'a'


    d.*ptr=10;

    d.print();


    dp->*ptr=20;

    dp->print();

}
```

Copy

```
a is 10

a is 20
```

The syntax is very tough, hence they are only used under special circumstances.

Pointer to Member Functions of Class

Pointers can be used to point to class's Member functions.

Syntax:

```
return_type (class_name::*ptr_name) (argument_type) =
&class_name::function_name;
```

Copy

Below is an example to show how we use ppointer to member functions.

```
class Data

{

    public:

    int f(float)

    {
        return 1;

    }

};


int (Data::*fp1) (float) = &Data::f;   // Declaration and assignment

int (Data::*fp2) (float);        // Only Declaration


int main(0

{

    fp2 = &Data::f;   // Assignment inside main()

}
```
Copy

Some Points to remember

1. You can change the value and behaviour of these pointers on runtime. That means, you can point it to other member function or member variable.
2. To have pointer to data member and member functions you need to make them public.

Inheritance in C++

Inheritance is the capability of one class to acquire properties and characteristics from another class. The class whose properties are inherited by other class is called the **Parent** or **Base** or **Super** class. And, the class which inherits properties of other class is called **Child** or **Derived** or **Sub** class.

Inheritance makes the code reusable. When we inherit an existing class, all its methods and fields become available in the new class, hence code is reused.

NOTE: All members of a class except Private, are inherited

Purpose of Inheritance in C++

1. Code Reusability
2. Method Overriding (Hence, Runtime Polymorphism.)
3. Use of Virtual Keyword

Basic Syntax of Inheritance

```
class Subclass_name : access_mode Superclass_name
```

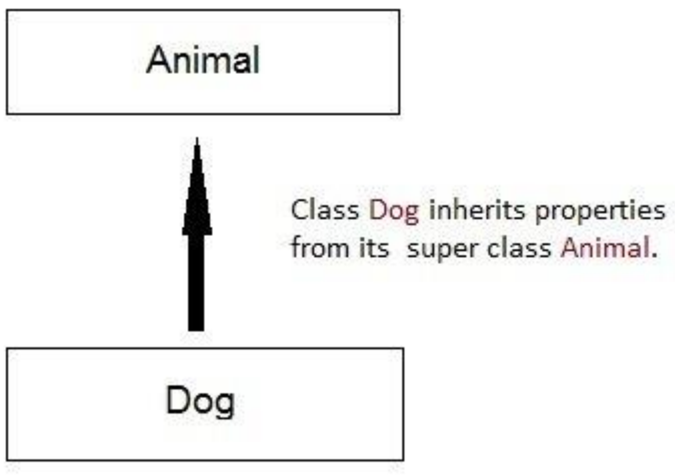
Copy

While defining a subclass like this, the super class must be already defined or atleast declared before the subclass declaration.

Access Mode is used to specify, the mode in which the properties of superclass will be inherited into subclass, public, privtate or protected.

Example of Inheritance

Whenever we want to use something from an existing class in a new class, we can use the concept on Inheritace. Here is a simple example,



```
class Animal

{

    public:

    int legs = 4;

};



// Dog class inheriting Animal class

class Dog : public Animal

{

    public:

    int tail = 1;

};



int main()
```

```
{

    Dog d;

    cout << d.legs;

    cout << d.tail;

}
```

Copy

```
4 1
```

Access Modifiers and Inheritance: Visibility of Class Members

Depending on Access modifier used while inheritance, the availability of class members of Super class in the sub class changes. It can either be private, protected or public.

1) Public Inheritance

This is the most used inheritance mode. In this the protected member of super class becomes protected members of sub class and public becomes public.

```
class Subclass : public Superclass
```

Copy

2) Private Inheritance

In private mode, the protected and public members of super class become private members of derived class.

```
class Subclass : Superclass    // By default its private inheritance
```

Copy

3) Protected Inheritance

In protected mode, the public and protected members of Super class becomes protected members of Sub class.

```
class subclass : protected Superclass
```

Copy

Table showing all the Visibility Modes

	Derived Class	**Derived Class**	**Derived Class**
Base class	**Public Mode**	**Private Mode**	**Protected Mode**
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

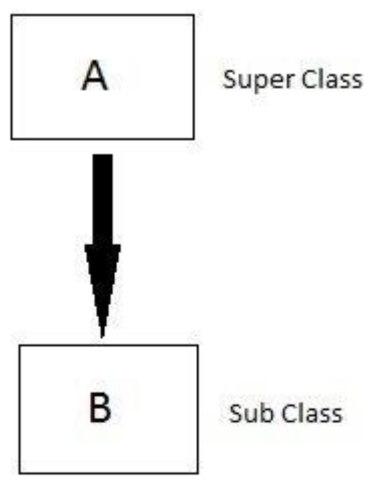
Types of Inheritance in C++

In C++, we have 5 different types of Inheritance. Namely,

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance (also known as Virtual Inheritance)

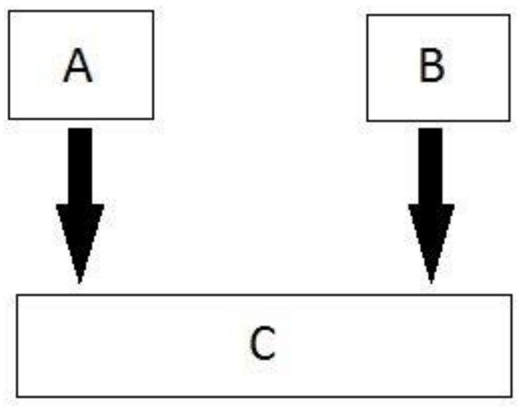
Single Inheritance in C++

In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of Inheritance.



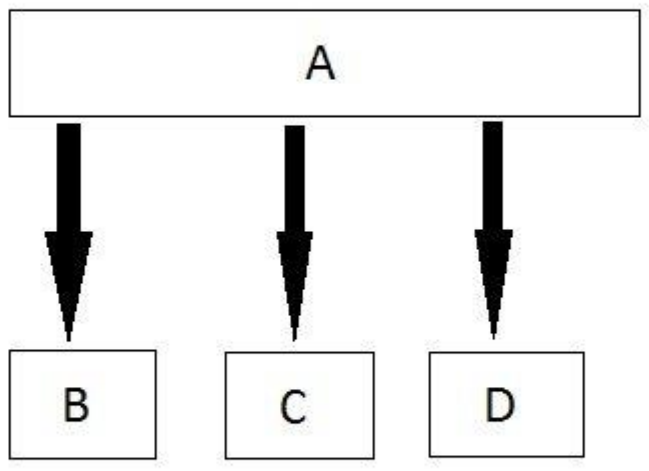
Multiple Inheritance in C++

In this type of inheritance a single derived class may inherit from two or more than two base classes.



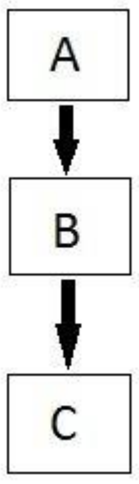
Hierarchical Inheritance in C++

In this type of inheritance, multiple derived classes inherits from a single base class.



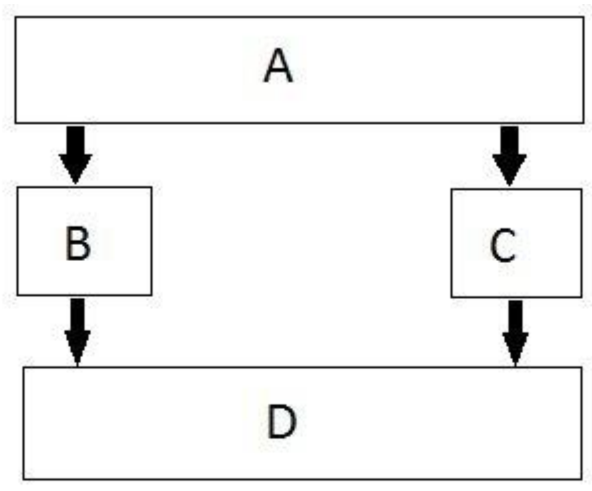
Multilevel Inheritance in C++

In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



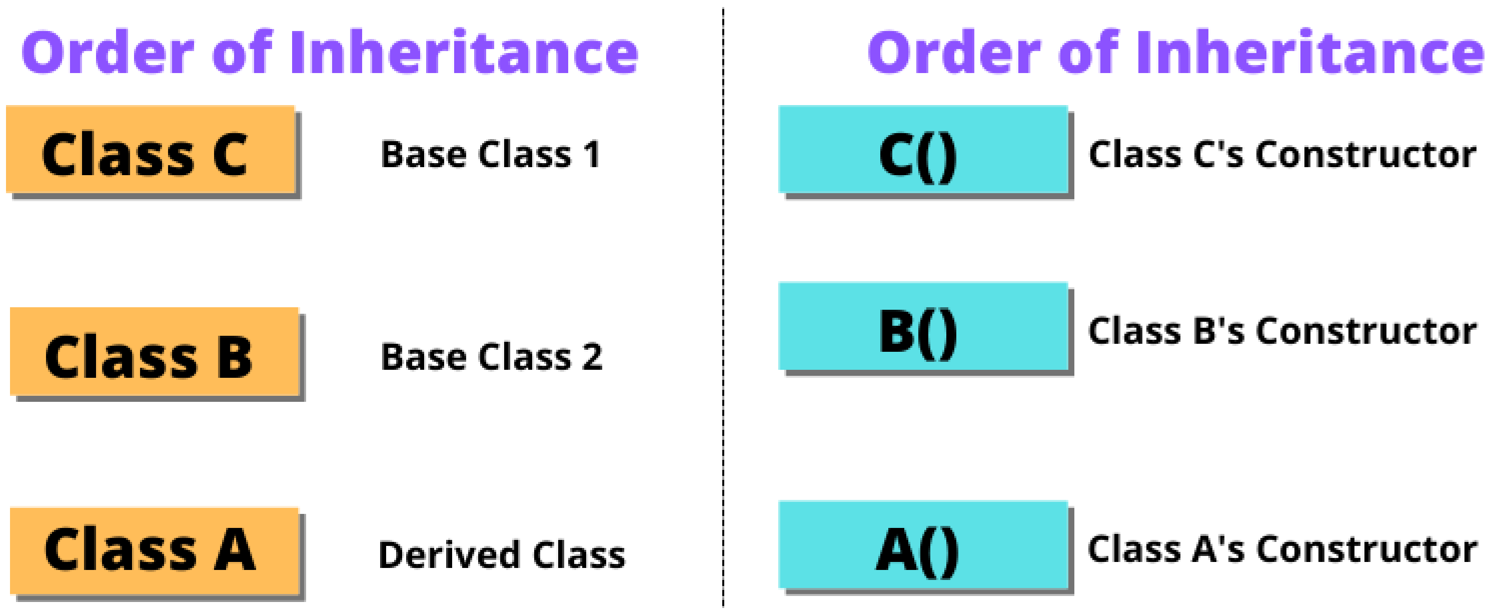
Hybrid (Virtual) Inheritance in C++

Hybrid Inheritance is combination of Hierarchical and Mutilevel Inheritance.



Order of Constructor Call with Inheritance in C++

In this tutorial, we will learn about the **Order of Constructor Call with Inheritance in C++**. If you are not familiar with the Constructor in C++, you can learn about it from C++ Constructors tutorial.



Base class Default Constructor in Derived class Constructors:

When we derive a class from the base class then all the data members of the base class will become a member of the derived class. We use the constructor to initialize the data members and here the obvious case is when the data is inherited into the derived class who will be responsible to initialize them? To initialize the inherited data membres constructor is necessary and that's why the constructor of the base class is called first. In the program given below, we can see the sequence of execution of constructors in inheritance is given below:

```
#include <iostream>

using namespace std;

class Base

{

   int x;



public:

   // default constructor
```

```
    Base()
    {
        cout << "Base default constructor\n";
    }
};

class Derived : public Base
{
    int y;

public:
    // default constructor
    Derived()
    {
        cout << "Derived default constructor\n";
    }
    // parameterized constructor
    Derived(int i)
    {
        cout << "Derived parameterized constructor\n";
    }
};

int main()
{
    Base b;
    Derived d1;
    Derived d2(10);
}
```
Copy

```
Base default constructor

Base default constructor

Derived default constructor

Base default constructor

Derived parameterized constructor
```

Base class Parameterized Constructor in Derived class Constructor:

Let's see how we can call the parameterized constructor in the Derived class, We need to explicitly define the calling for the parameterized constructor of the derived class using `:` operator while defining the parameterized constructor in a derived class.

```
#include <iostream>

using namespace std;

class Base

{

    int x;

    public:

    // parameterized constructor

    Base(int i)

    {

        x = i;

        cout << "Base Parameterized Constructor\n";

    }

};


class Derived: public Base

{

    int y;

    public:

    // parameterized constructor

    Derived(int j):Base(j)

    {

        y = j;

        cout << "Derived Parameterized Constructor\n";
    }

};



int main()

{

    Derived d(10) ;

}
```

Copy

```
Base Parameterized Constructor

Derived Parameterized Constructor
```

Here are some basic rules to figure out the **Order of Constructor Call with Inheritance in C++**.

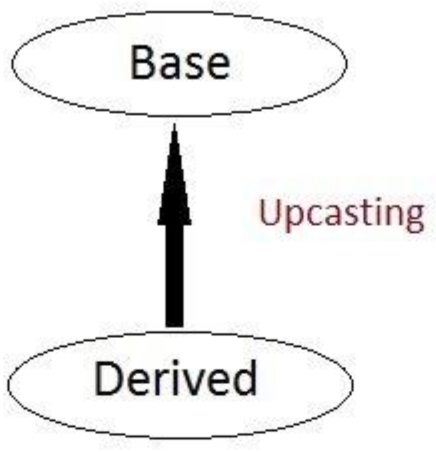
- Construction always starts with the base class. If there are multiple base classes then, construction starts with the leftmost base. If there is a virtual inheritance then it's given higher preference).
- Then the member fields are constructed. They are initialized in the order they are declared
- Finally, the class itself is constructed
- The order of the destructor is exactly the reverse

Related Tutorials:

- OOPS Basic Concepts
- Introduction to C++
- Functions in C++
- References in C++

Upcasting in C++

Upcasting is using the Super class's reference or pointer to refer to a Sub class's object. Or we can say that, the act of converting a Sub class's reference or pointer into its Super class's reference or pointer is called Upcasting.



```
class Super

{
    int x;

    public:

    void funBase()

    {

        cout << "Super function";

    }

};


class Sub:public Super

{

    int y;

};


int main()

{

    Super* ptr;    // Super class pointer

    Sub obj;

    ptr = &obj;


    Super &ref;    // Super class's reference

    ref=obj;

}
```

Copy

The opposite of Upcasting is **Downcasting**, in which we convert Super class's reference or pointer into derived class's reference or pointer. We will study more about Downcasting later

---

Functions that are never Inherited

- Constructors and Destructors are never inherited and hence never overrided.(We will study the concept of function overriding in the next tutorial)
- Also, assignment operator `=` is never inherited. It can be overloaded but can't be inherited by sub class.

Inheritance and Static Functions in C++

1. They are inherited into the derived class.
2. If you redefine a static member function in derived class, all the other overloaded functions in base class are hidden.
3. Static Member functions can never be virtual. We will study about Virtual in coming topics.

Hybrid Inheritance and Virtual Class in C++

In Multiple Inheritance, the derived class inherits from more than one base class. Hence, in Multiple Inheritance there are a lot chances of ambiguity.

```
class A

{

    void show();

};


class B:public A

{

    // class definition

};


class C:public A

{

    // class defintion

};


class D:public B, public C

{

    // class definition

};


int main()

{
    D obj;

    obj.show();

}
```

Copy
In this case both class B and C inherits function show() from class A. Hence class D has two inherited copies of function show(). In main() function when we call function show(), then ambiguity arises, because compiler doesn't know which show() function to call. Hence we use **Virtual** keyword while inheriting class.

```
class B : virtual public A

{

    // class definition

};


class C : virtual public A

{

    // class definition

};


class D : public B, public C

{

    // class definition

};
```

Copy
Now by adding virtual keyword, we tell compiler to call any one out of the two show() funtions.

Hybrid Inheritance and Constructor call

As we all know that whenever a derived class object is instantiated, the base class constructor is always called. But in case of Hybrid Inheritance, as discussed in above example, if we create an instance of class D, then following constructors will be called :

- before class D's constructor, constructors of its super classes will be called, hence constructors of class B, class C and class A will be called.
- when constructors of class B and class C are called, they will again make a call to their super class's constructor.

This will result in multiple calls to the constructor of class A, which is undesirable. As there is a single instance of virtual base class which is shared by multiple classes that inherit from it, hence the constructor of the base class is only called once by the constructor of concrete class, which in our case is class D.

If there is any call for initializing the constructor of class A in class B or class C, while creating object of class D, all such calls will be skipped.

Polymorphism and Method Overriding in C++

In this tutorial we will cover the concepts of Polymorphism in C++ and Function overriding in C++. We will also see both of these in action using simple code examples.

Polymorphism in C++

Polymorphism means having multiple forms of one thing. In inheritance, polymorphism is done, by method overriding, when both super and sub class have member function with same declaration bu different definition.

Method Overriding in C++

If we inherit a class into the derived class and provide a definition for one of the base class's function again inside the derived class, then that function is said to be **overridden**, and this mechanism is called **Function Overriding**

Requirements for Overriding a Function

1. Inheritance should be there. Function overriding cannot be done within a class. For this we require a derived class and a base class.
2. Function that is redefined must have exactly the same declaration in both base and derived class, that means same name, same return type and same parameter list.

Example of Function Overriding in C++

```
class Base

{

    public:

    void show()

    {

        cout << "Base class";

    }

};
```

```
class Derived:public Base

{

    public:

    void show()

    {

        cout << "Derived Class";

    }

}
```

Copy

In this example, function show() is overridden in the derived class. Now let us study how these overridden functions are called in main() function.

Function Call Binding with class Objects

Connecting the function call to the function body is called **Binding**. When it is done before the program is run, its called **Early** Binding or **Static** Binding or **Compile-time** Binding.

```
class Base

{

    public:

    void shaow()

    {

        cout << "Base class\n";

    }

};



class Derived:public Base

{

    public:

    void show()

    {

        cout << "Derived Class\n";

    }
}


int main()

{

    Base b;       //Base class object

    Derived d;     //Derived class object

    b.show();     //Early Binding Ocuurs

    d.show();

}
```

Copy

```
Base class

Derived class
```

In the above example, we are calling the overrided function using Base class and Derived class object. Base class object will call base version of the function and derived class's object will call the derived version of the function.

Function Call Binding using Base class Pointer

But when we use a Base class's pointer or reference to hold Derived class's object, then Function call Binding gives some unexpected results.

```
class Base

{

    public:

    void show()

    {

        cout << "Base class\n";

    }

};


class Derived:public Base

{

    public:

    void show()
    {

        cout << "Derived Class\n";

    }

}


int main()

{

    Base* b;        //Base class pointer

    Derived d;      //Derived class object

    b = &d;

    b->show();      //Early Binding Occurs

}
```

Copy

```
Base class
```

In the above example, although, the object is of Derived class, still Base class's method is called. This happens due to Early Binding.

Compiler on seeing **Base class's pointer**, set call to Base class's `show()` function, without knowing the actual object type.

Virtual Functions in C++

Virtual Function is a function in base class, which is overrided in the derived class, and which tells the compiler to perform **Late Binding** on this function.

`Virtual` Keyword is used to make a member function of the base class Virtual.

Late Binding in C++

In Late Binding function call is resolved at runtime. Hence, now compiler determines the type of object at runtime, and then binds the function call. Late Binding is also called **Dynamic** Binding or **Runtime** Binding.

Problem without Virtual Keyword

Let's try to understand what is the issue that `virtual` keyword fixes,

```
class Base

{
    public:

    void show()

    {

        cout << "Base class";

    }

};


class Derived:public Base

{

    public:

    void show()

    {

        cout << "Derived Class";

    }

}


int main()

{

    Base* b;       //Base class pointer

    Derived d;     //Derived class object

    b = &d;

    b->show();     //Early Binding Ocuurs

}
```

Copy

```
Base class
```

When we use Base class's pointer to hold Derived class's object, base class pointer or reference will always call the base version of the function

---

Using Virtual Keyword in C++

We can make base class's methods virtual by using **virtual** keyword while declaring them. Virtual keyword will lead to Late Binding of that method.

```
class Base
{
    public:
    virtual void show()
    {
        cout << "Base class\n";
    }
};

class Derived:public Base
{
    public:
    void show()
    {
        cout << "Derived Class";
    }
}

int main()
{
    Base* b;        //Base class pointer
    Derived d;      //Derived class object
    b = &d;
    b->show();      //Late Binding Ocuurs
}
```

Copy

```
Derived class
```

On using Virtual keyword with Base class's function, Late Binding takes place and the derived version of function will be called, because base class pointer pointes to Derived class object.

Using Virtual Keyword and Accessing Private Method of Derived class

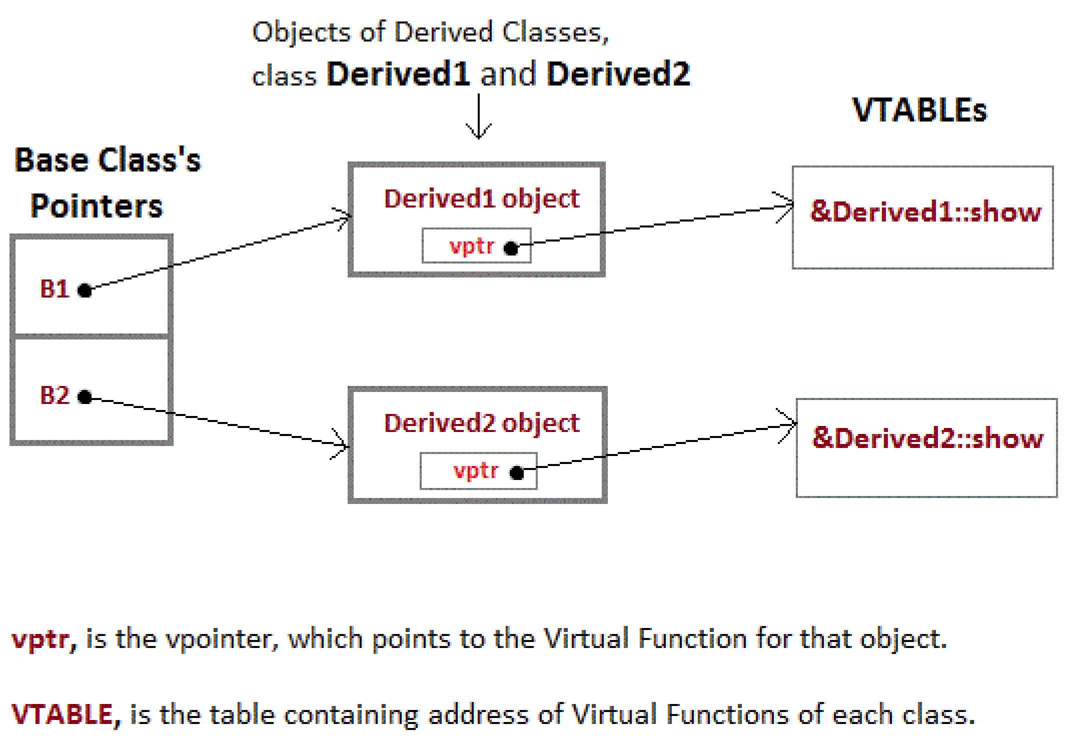
We can call **private** function of derived class from the base class pointer with the help of virtual keyword. Compiler checks for access specifier only at compile time. So at run time when late binding occurs it does not check whether we are calling the private function or public function.

```
#include <iostream>

using namespace std;


class A

{

    public:

    virtual void show()

    {

        cout << "Base class\n";

    }

};


class B: public A

{

    private:

    virtual void show()

    {

        cout << "Derived class\n";

    }

};


int main()

{

    A *a;

    B b;

    a = &b;

    a->show();

}
```

Copy

```
Derived class
```

Mechanism of Late Binding in C++


Objects of Derived Classes,
class **Derived1** and **Derived2**
VTABLEs
Base Class's Pointers
B1
B2
Derived1 object
vptr
Derived2 object
vptr
&Derived1::show
&Derived2::show
vptr, is the vpointer, which points to the Virtual Function for that object.
VTABLE, is the table containing address of Virtual Functions of each class.

To accomplich late binding, Compiler creates **VTABLEs**, for each class with virtual function. The address of virtual functions is inserted into these tables. Whenever an object of such class is created the compiler secretly inserts a pointer called **vpointer**, pointing to VTABLE for that object. Hence when function is called, compiler is able to resovle the call by binding the correct function using the vpointer.

Important Points to Remember

1. Only the Base class Method's declaration needs the **Virtual** Keyword, not the definition.
2. If a function is declared as **virtual** in the base class, it will be virtual in all its derived classes.
3. The address of the virtual Function is placed in the **VTABLE** and the copiler uses **VPTR**(vpointer) to point to the Virtual Function.

Abstract Class and Pure Virtual Function in C++

Abstract Class is a class which contains atleast one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristics of Abstract Class

1. Abstract class cannot be instantiated, but pointers and refrences of Abstract class type can be created.
2. Abstract class can have normal functions and variables along with a pure virtual function.
3. Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
4. Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Pure Virtual Functions in C++

Pure virtual Functions are virtual functions with no definition. They start with **virtual** keyword and ends with `= 0`. Here is the syntax for a pure virtual function,

```
virtual void f() = 0;
```

Copy

Example of Abstract Class in C++

Below we have a simple example where we have defined an abstract class,

```
//Abstract base class

class Base

{

    public:

    virtual void show() = 0;     // Pure Virtual Function

};



class Derived:public Base

{

    public:

    void show()

    {

        cout << "Implementation of Virtual Function in Derived 
class\n";

    }

};
```

```
int main()

{

    Base obj;   //Compile Time Error

    Base *b;

    Derived d;

    b = &d;

    b->show();

}
```

Copy

```
Implementation of Virtual Function in Derived class
```

In the above example Base class is abstract, with pure virtual **show()** function, hence we cannot create object of base class.

Why can't we create Object of an Abstract Class?

When we create a pure virtual function in Abstract class, we reserve a slot for a function in the **VTABLE**(studied in last topic), but doesn't put any address in that slot. Hence the VTABLE will be incomplete.

As the VTABLE for Abstract class is incomplete, hence the compiler will not let the creation of object for such class and will display an errror message whenever you try to do so.

Pure Virtual definitions

- Pure Virtual functions can be given a small definition in the Abstract class, which you want all the derived classes to have. Still you cannot create object of Abstract class.
- Also, the Pure Virtual function must be defined outside the class definition. If you will define it inside the class definition, complier will give an error. Inline pure virtual definition is Illegal.

```
// Abstract base class

class Base

{

    public:

    virtual void show() = 0;     //Pure Virtual Function

};



void Base :: show()     //Pure Virtual definition
```

```
{
    cout << "Pure Virtual definition\n";
}


class Derived:public Base
{
    public:
    void show()
    {
        cout << "Implementation of Virtual Function in Derived 
class\n";
    }
};


int main()
{
    Base *b;
    Derived d;
    b = &d;
    b->show();
}
```

Copy

```
Implementation of Virtual Function in Derived class
```

Virtual Destructors in C++

Destructors in the Base class can be Virtual. Whenever Upcasting is done, Destructors of the Base class must be made virtual for proper destrucstion of the object when the program exits.

NOTE: Constructors are never Virtual, only Destructors can be Virtual.

Upcasting without Virtual Destructor in C++

Lets first see what happens when we do not have a virtual Base class destructor.

```
class Base

{

    public:

    ~Base()

    {

        cout << "Base Destructor\n";

    }

};


class Derived:public Base

{

    public:

    ~Derived()

    {

        cout<< "Derived Destructor\n";

    }

};


int main()

{

    Base* b = new Derived;      // Upcasting

    delete b;

}
```

Copy

```
Base Destructor
```

In the above example, `delete b` will only call the Base class destructor, which is undesirable because, then the object of Derived class remains undestructed, because its destructor is never called. Which results in memory leak.

Upcasting with Virtual Destructor in C++

Now lets see. what happens when we have Virtual destructor in the base class.

```
class Base
{
    public:
    virtual ~Base()
    {
        cout << "Base Destructor\n";
    }
};

class Derived:public Base
{
    public:
    ~Derived()
    {
        cout<< "Derived Destructor";
    }
};

int main()
{
    Base* b = new Derived;     // Upcasting
    delete b;
}
```

Copy

```
Derived Destructor
Base Destructor
```

When we have Virtual destructor inside the base class, then first Derived class's destructor is called and then Base class's destructor is called, which is the desired behaviour.

Pure Virtual Destructors in C++

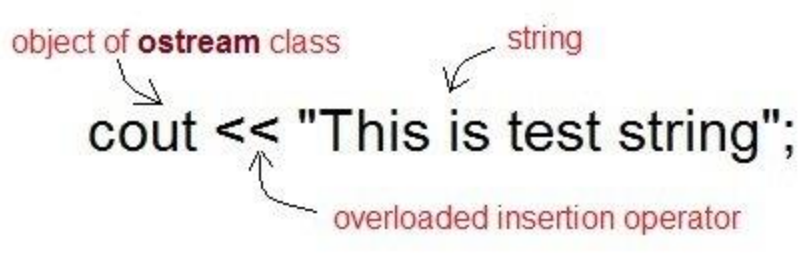
- Pure Virtual Destructors are legal in C++. Also, pure virtual Destructors must be defined, which is against the pure virtual behaviour.
- The only difference between Virtual and Pure Virtual Destructor is, that pure virtual destructor will make its Base class Abstract, hence you cannot create object of that class.
- There is no requirement of implementing pure virtual destructors in the derived classes.

```
class Base

{

    public:

    virtual ~Base() = 0;     // Pure Virtual Destructor

};


// Definition of Pure Virtual Destructor

Base::~Base()

{

    cout << "Base Destructor\n";

}


class Derived:public Base

{

    public:

    ~Derived()

    {

        cout<< "Derived Destructor";

    }

};
```

Copy

Operator Overloading in C++

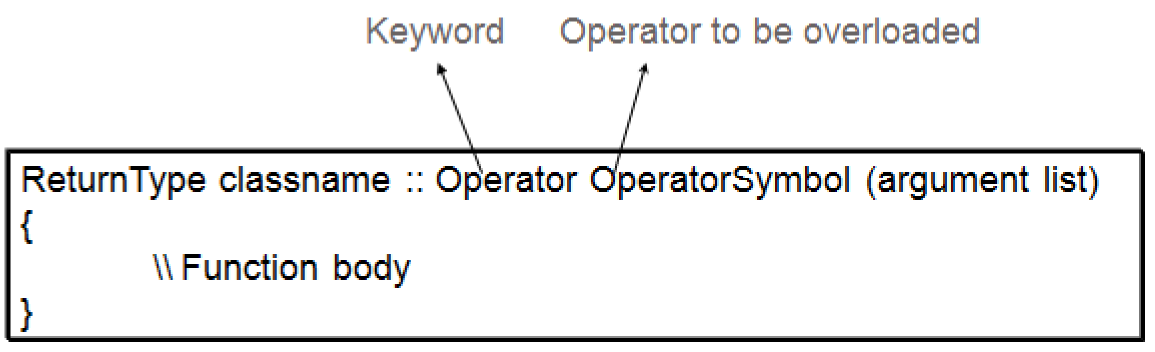
Operator overloading is an important concept in C++. It is polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type. For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String(concatenation) etc.



Almost any operator can be overloaded in C++. However there are few operator which can not be overloaded. **Operator that are not overloaded** are follows

- scope operator - ::
- sizeof
- member selector - .
- member pointer selector - *
- ternary operator - ?:

Operator Overloading Syntax



```
ReturnType classname :: Operator OperatorSymbol (argument list)
{
        \\ Function body
}
```

Implementing Operator Overloading in C++

Operator overloading can be done by implementing a function which can be :

1. Member Function
2. Non-Member Function
3. Friend Function

Operator overloading function can be a member function if the Left operand is an Object of that class, but if the Left operand is different, then Operator overloading function must be a non-member function.

Operator overloading function can be made friend function if it needs access to the private and protected members of class.

Restrictions on Operator Overloading in C++

Following are some restrictions to be kept in mind while implementing operator overloading.

1. Precedence and Associativity of an operator cannot be changed.

2. Arity (numbers of Operands) cannot be changed. Unary operator remains unary, binary remains binary etc.
3. No new operators can be created, only existing operators can be overloaded.
4. Cannot redefine the meaning of a procedure. You cannot change how integers are added.

Operator Overloading Examples in C++

Almost all the operators can be overloaded in infinite different ways. Following are some examples to learn more about operator overloading. All the examples are closely connected.

Overloading Arithmetic Operator in C++

Arithmetic operator are most commonly used operator in C++. Almost all arithmetic operator can be overloaded to perform arithmetic operation on user-defined data type. In the below example we have overridden the **+** operator, to add to Time(hh:mm:ss) objects.

Example: overloading + Operator to add two Time class object

```
#include <iostream.h>

#include <conio.h>

class Time

{

    int h,m,s;

    public:

    Time()

    {

        h=0, m=0; s=0;

    }

    void setTime();

    void show()

    {

        cout<< h<< ":"<< m<< ":"<< s;

    }


    //overloading '+' operator

    Time operator+(time);
```

```cpp
};


Time Time::operator+(Time t1)  //operator function

{

    Time t;

    int a,b;

    a = s+t1.s;

    t.s = a%60;

    b = (a/60)+m+t1.m;

    t.m = b%60;

    t.h = (b/60)+h+t1.h;

    t.h = t.h%12;

    return t;

}


void time::setTime()

{

    cout << "\n Enter the hour(0-11)  ";

    cin >> h;

    cout << "\n Enter the minute(0-59)  ";

    cin >> m;

    cout << "\n Enter the second(0-59)  ";

    cin >> s;

}


void main()

{

    Time t1,t2,t3;


    cout << "\n Enter the first time ";
    t1.setTime();

    cout << "\n Enter the second time ";

    t2.setTime();

    t3 = t1 + t2;    //adding of two time object using '+' operator

    cout << "\n First time ";

    t1.show();

    cout << "\n Second time ";

    t2.show();

    cout << "\n Sum of times ";

    t3.show();

    getch();

}
```

Copy
While normal addition of two numbers return the sumation result. In the case above we have overloaded the `+` operator, to perform addition of two `Time` class objects. We add the **seconds**, **minutes** and **hour** values separately to return the new value of time.

In the `setTime()` funtion we are separately asking the user to enter the values for hour, minute and second, and then we are setting those values to the `Time` class object.

For inputs, `t1` as **01:20:30**(1 hour, 20 minute, 30 seconds) and `t2` as **02:15:25**(2 hour, 15 minute, 25 seconds), the output for the above program will be:

```
1:20:30

2:15:25

3:35:55
```

First two are values of `t1` and `t2` and the third is the result of their addition.

Overloading I/O operator in C++

The first question before learning how to override the I/O operator should be, why we need to override the I/O operators. Following are a few cases, where overloading the I/O operator proves useful:

- We can overload output operator `<<` to print values for user defined datatypes.
- We can overload output operator `>>` to input values for user defined datatypes.

In case of input/output operator overloading, **left operand** will be of types `ostream&` and `istream&`

Also, when overloading these operators, we must make sure that the functions must be a Non-Member function because left operand is not an object of the class.

And it must be a **friend function** to access private data members.

You have seen above that `<<` operator is overloaded with **ostream** class object `cout` to print primitive datatype values to the screen, which is the default behavious of `<<` operator, when used with `cout`. In other words, it is already overloaded by default.

Similarly we can overload `<<` operator in our class to print user-defined datatypes to screen. For example we can overload `<<` in our `Time` class to display the value of `Time` object using `cout` rather than writing a custom member function like `show()` to print the value of `Time` class objects.

```
Time t1(3,15,48);

// like this, directly

cout << t1;
```

Copy
In the next section we will learn how to do that.

NOTE: When the operator does not modify its operands, the best way to overload the operator is via friend function.

Example: overloading `<<` Operator to print Class Object

We will now overload the `<<` operator in the `Time` class,

```
#include<iostream.h>

#include<conio.h>

class Time

{

    int hr, min, sec;

    public:

    // default constructor

    Time()

    {

        hr=0, min=0; sec=0;

    }


    // overloaded constructor

    Time(int h, int m, int s)

    {

        hr=h, min=m; sec=s;
    }


    // overloading '<<' operator

    friend ostream& operator << (ostream &out, Time &tm);

};


// define the overloaded function

ostream& operator << (ostream &out, Time &tm)

{

    out << "Time is: " << tm.hr << " hour : " << tm.min << " min : " <<
tm.sec << " sec";

    return out;

}


void main()

{

    Time tm(3,15,45);

    cout << tm;

}
```

Copy

```
Time is: 3 hour : 15 min : 45 sec
```

This is simplied in languages like Core Java, where all you need to do in a class is override the `toString()` method of the `String` class, and you can define how to print the object of that class.

Overloading Relational Operator in C++

You can also overload relational operators like `==` , `!=` , `>=` , `<=` etc. to compare two object of any class.

Let's take a quick example by overloading the `==` operator in the `Time` class to directly compare two objects of `Time` class.

```
class Time

{

    int hr, min, sec;
```

```cpp
    public:

    // default constructor

    Time()

    {

        hr=0, min=0; sec=0;

    }


    // overloaded constructor

    Time(int h, int m, int s)

    {

        hr=h, min=m; sec=s;

    }


    //overloading '==' operator

    friend bool operator==(Time &t1, Time &t2);

};


/*

    Defining the overloading operator function

    Here we are simply comparing the hour, minute and

    second values of two different Time objects to compare

    their values

*/

bool operator== (Time &t1, Time &t2)

{

    return ( t1.hr == t2.hr && t1.min == t2.min && t1.sec == t2.sec );

}


void main()

{
    Time t1(3,15,45);

    Time t2(4,15,45);

    if(t1 == t2)

    {

        cout << "Both the time values are equal";

    }

    else

    {

        cout << "Both the time values are not equal";

    }

}
```

Copy

```
Both the time values are not equal
```

As the hour value of object t1 is **3** and for object t2 it is **4**, hence they are not equal.

Copy Constructor vs. Assignment Operator (=)

Assignment operator is used to copy the values from one object to another **already existing object**. For example:

```
Time tm(3,15,45); // tm object created and initialized

Time t1;     // t1 object created

t1 = tm;     // initializing t1 using tm
```

Copy

Whereas, **Copy constructor** is a special constructor that initializes a **new object** from an existing object.

```
Time tm(3,15,45); // tm object created and initialized

Time t1(tm);     //t1 object created and initialized using tm object
```

Copy

In case of Copy constructor, we provide the object to be copied as an argument to the constructor. Also, we first need to define a copy constructor in our class. We have covered Copey constructor in detail here: [Copy Constructor in C++]
