UNIT -3

- **Constructors in C++**

Constructor is a special method that is invoked automatically at the time of object creation. It is used to initialize the data members of new objects generally. The constructor in C++ has the same name as the class or structure. Constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object which is why it is known as constructors.

Constructor does not have a return value, hence they do not have a return type.

The prototype of Constructors is as follows:

<class-name> (list-of-parameters);

Constructors can be defined inside or outside the class declaration:-

The syntax for defining the constructor within the class:

<class-name> (list-of-parameters) { // constructor definition }

The syntax for defining the constructor outside the class:

<class-name>: :<class-name> (list-of-parameters){ // constructor definition}

Example:

// defining the constructor within the class

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

class student {
int rno;
char name[10];
double fee;

public:
student()
{
cout << "Enter the RollNo:";
cin >> rno;
cout << "Enter the Name:";
cin >> name;
cout << "Enter the Fee:";
cin >> fee;
}

void display()
{
cout << endl << rno << "\t" << name << "\t" << fee;
}
};

int main()
{
   student s; // constructor gets called automatically when
              // we create the object of the class
   s.display();

   return 0;
}
```

Output:

```
Enter the RollNo:Enter the Name:Enter the Fee:
0       6.95303e-310
```

// defining the constructor outside the class

```
#include <iostream>
using namespace std;
class student {
   int rno;
   char name[50];
   double fee;

public:
   student();
   void display();
};

student::student()
{
   cout << "Enter the RollNo:";
```

```
        cin >> rno;

        cout << "Enter the Name:";
        cin >> name;

        cout << "Enter the Fee:";
        cin >> fee;
    }

    void student::display()
    {
        cout << endl << rno << "\t" << name << "\t" << fee;
    }

    int main()
    {
        student s;
        s.display();

        return 0;
    }
```

Output:

```
Enter the RollNo: 30
Enter the Name: ram
Enter the Fee: 20000
30 ram 20000
```

A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself.
- Default Constructors don’t have input argument however, Copy and Parameterized Constructors have input arguments.
- Constructors don’t have return type.
- A constructor is automatically called when an object is created.
- It must be placed in public section of class.
- If we do not specify a constructor, C++ compiler generates a default constructor for object (expects no parameters and has an empty body).

Characteristics of the constructor:

- The name of the constructor is the same as its class name.
- Constructors are mostly declared in the public section of the class though it can be declared in the private section of the class.
- Constructors do not return values; hence they do not have a return type.
- A constructor gets called automatically when we create the object of the class.
- Constructors can be overloaded.
- Constructor cannot be declared virtual.
- Constructor cannot be inherited.
- Addresses of Constructor cannot be referred.
- Constructors makes implicit calls to new and delete operators during memory allocation.

➢ **Types of Constructors:**

1. **Default Constructors:** Default constructor is the constructor which doesn't take any argument. It has no parameters. It is also called a zero-argument constructor.

Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.

```
// Cpp program to illustrate the
// concept of Constructors
#include <iostream>
using namespace std;

class construct {
public:
    int a, b;

    // Default Constructor
    construct()
    {
        a = 10;
        b = 20;
    }
};

int main()
{
    // Default constructor called automatically
    // when the object is created
    construct c;
    cout << "a: " << c.a << endl << "b: " << c.b;
    return 1;
}
```

Output:

```
a: 10
b: 20
```

2. **Parameterized Constructors:** It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

When the parameterized constructor is defined and no default constructor is defined explicitly, the compiler will not implicitly call the default constructor and hence creating a simple object as

```
Student s;
```

Will flash an error

```
// CPP program to illustrate
// parameterized constructors
#include <iostream>
using namespace std;

class Point {
private:
    int x, y;

public:
    // Parameterized Constructor
    Point(int x1, int y1)
    {
        x = x1;
        y = y1;
    }
    int getX() { return x; }
    int getY() { return y; }
};

int main()
{
// Constructor called
    Point p1(10, 15);

    // Access values assigned by constructor
    cout << "p1.x = " << p1.getX()
         << ", p1.y = " << p1.getY();

    return 0;
}
```

Output :

```
p1.x = 10, p1.y = 15
```

When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly.

Example e = Example(0, 50); // **Explicit call**

Example e(0, 50);   // **Implicit call**

- **Uses of Parameterized constructor:**
  - It is used to initialize the various data elements of different objects with different values when they are created.
  - It is used to overload constructors.

3. Copy Constructor:

A copy constructor is a member function that initializes an object using another object of the same class.

Whenever we define one or more non-default constructors (with parameters) for a class, a default constructor (without parameters) should also be explicitly defined as the compiler will not provide a default constructor in this case. However, it is not necessary but it’s considered to be the best practice to always define a default constructor.

Copy constructor takes a reference to an object of the same class as an argument.

Example:

```
// C++ program to demonstrate the working
// of a COPY CONSTRUCTOR
#include <iostream>
using namespace std;

class Point {
private:
    int x, y;

public:
    Point(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

    // Copy constructor
    Point(const Point& p1)
    {
        x = p1.x;
        y = p1.y;
        }

    int getX() { return x; }
    int getY() { return y; }
};

int main()
{
    Point p1(10, 15); // Normal constructor is called here
    Point p2 = p1; // Copy constructor is called here

    // Let us access values assigned by constructors
    cout << "p1.x = " << p1.getX()
         << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX()
```

```
        << ", p2.y = " << p2.getY();
    return 0;
}
```

Output:

```
p1.x = 10, p1.y = 15
p2.x = 10, p2.y = 15
```

- **Destructor:**

A destructor is also a special member function as a constructor. Destructor destroys the class objects created by the constructor. Destructor has the same name as their class name preceded by a tilde (~) symbol. It is not possible to define more than one destructor. The destructor is only one way to destroy the object created by the constructor. Hence destructor can-not be overloaded. Destructor neither requires any argument nor returns any value. It is automatically called when the object goes out of scope. Destructors release memory space occupied by the objects created by the constructor. In destructor, objects are destroyed in the reverse of object creation.

The syntax for defining the destructor within the class

```
~ <class-name>()
{
 }
```

The syntax for defining the destructor outside the class

```
<class-name>: : ~ <class-name>(){}
```

Example:

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

class Test {
public:
    Test() { cout << "\n Constructor executed"; }

    ~Test() { cout << "\n Destructor executed"; }
};
```

```
main()
{
   Test t;
   return 0;
}
```

Output:

```
 Constructor executed
 Destructor executed
```

Characteristics of a destructor:-

- Destructor is invoked automatically by the compiler when its corresponding constructor goes out of scope and releases the memory space that is no longer required by the program.
- Destructor neither requires any argument nor returns any value therefore it cannot be overloaded.
- Destructor cannot be declared as static and const;
- Destructor should be declared in the public section of the program.
- Destructor is called in the reverse order of its constructor invocation.

Constructor Overloading:

In C++, We can have more than one constructor in a class with same name, as long as each has a different list of arguments. This concept is known as **Constructor Overloading** and is quite similar to function overloading.

- Overloaded constructors essentially have the same name (exact name of the class) and different by number and type of arguments.
- A constructor is called depending upon the number and type of arguments passed.
- While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

Example:

```
// C++ program to illustrate
// Constructor overloading
#include <iostream>
using namespace std;
```

```
class construct
{

public:
    float area;

    // Constructor with no parameters
    construct()
    {
        area = 0;
    }
    // Constructor with two parameters
    construct(int a, int b)
    {
        area = a * b;
    }

    void disp()
    {
        cout<< area<< endl;
    }
};

int main()
{
    // Constructor Overloading
    // with two different constructors
    // of class name
    construct o;
    construct o2( 10, 20);

    o.disp();
    o2.disp();
    return 1;
}
```

Output:

```
0
200
```

- **Inheritance**

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. It is a feature that enables a class to acquire properties and characteristics of another class. Inheritance allows you to reuse your code since the derived class or the child class can reuse the members of the base class by inheriting them. Consider a real-life example to clearly understand the concept of inheritance. A child inherits some properties from his/her parents, such as the ability to speak, walk, and eat, and so on. But these properties are not especially inherited in his parents only. His parents inherit these properties from another class called mammals. This mammal class again derives these characteristics from the animal class. Inheritance works in the same manner. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

During inheritance, the data members of the base class get copied in the derived class and can be accessed depending upon the visibility mode used. The order of the accessibility is always in a decreasing order i.e., from public to protected. There are mainly five types of Inheritance in C++ that you will explore in this article. They are as follows:

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

- **Child and Parent classes :**

To understand the concept of Inheritance, two terms on which the whole concept of inheritance is based - Child class and Parent class.

Child class: The class that inherits the characteristics of another class is known as the child class or derived class. The number of child classes that can be inherited from a single parent class is based upon the type of inheritance. A child class will access the data members of the parent class according to the visibility mode specified during the declaration of the child class.

Parent class: The class from which the child class inherits its properties is called the parent class or base class. A single parent class can derive multiple child classes (Hierarchical Inheritance) or multiple parent classes can inherit a single base class (Multiple Inheritance). This depends on the different types of inheritance in C++.

The syntax for defining the child class and parent class in all types of Inheritance in C++ is given below:

```
class parent_class
{
   //class definition of the parent class
};
class child_class : visibility_mode parent_class
{
  //class definition of the child class
};
```

Syntax Description

- **parent_class:** Name of the base class or the parent class.
- **child_class:** Name of the derived class or the child class.
- **visibility_mode:** Type of the visibility mode (i.e., private, protected, and public) that specifies how the data members of the child class inherit from the parent class.

- **Uses of Inheritance:-**

Inheritance makes the programming more efficient and is used because of the benefits it provides. The most important usages of inheritance are discussed below:

- **Code reusability:** One of the main reasons to use inheritance is that you can reuse the code. For example, consider a group of animals as separate classes - Tiger, Lion, and Panther. For these classes, you can create member functions like the predator () as they all are predators, canine() as they all have canine teeth to hunt, and claws() as all the three animals have big and sharp claws. Now, since all the three functions are the same for these classes, making separate functions for all of them will cause data redundancy and can increase the chances of error. So instead of this, you can use inheritance here. You can create a base class named carnivores and add these functions to it and inherit these functions to the tiger, lion, and panther classes.
- **Transitive nature:** Inheritance is also used because of its transitive nature. For example, you have a derived class mammal that inherits its properties from the base class animal. Now, because of the transitive nature of the inheritance, all the child classes of 'mammal' will inherit the properties of the class 'animal' as well. This helps in debugging to a great extent. You can remove the bugs from your base class and all the inherited classes will automatically get debugged.

➢ **Modes of inheritance:**

There are three modes of inheritance:

- Public mode
- Protected mode
- Private mode

- **Public mode:**

In the public mode of inheritance, when a child class is derived from the base or parent class, then the public member of the base class or parent class will become public in the child class also, in the same way, the protected member of the base class becomes protected in the child class, and private members of the base class are not accessible in the derived class.

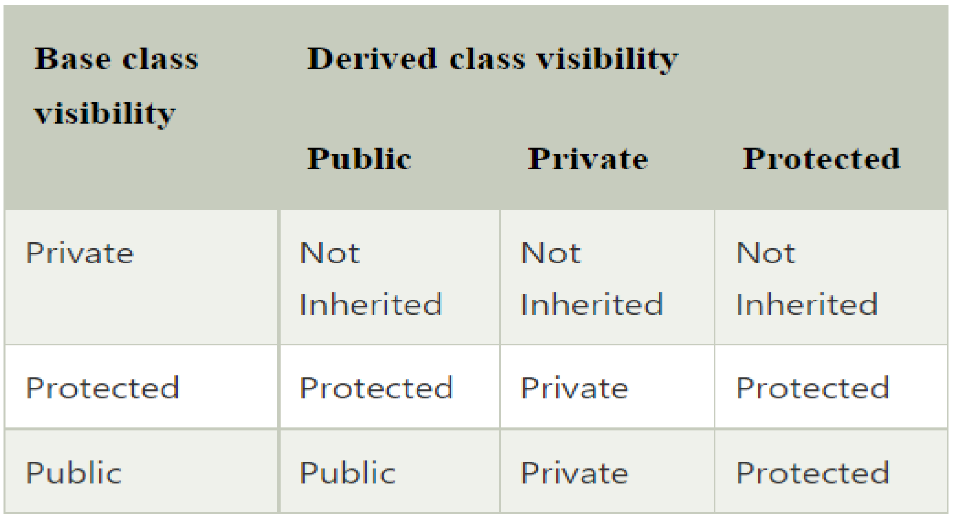
- **Protected mode:**

In protected mode, when a child class is derived from a base class or parent class, then both public and protected members of the base class will become protected in the derived class, and private members of the base class are again not accessible in the derived class. In contrast, protected members can be easily accessed in the derived class.

- **Private mode:**

In private mode, when a child class is derived from a base class, then both public and protected members of the base class will become private in the derived class, and private members of the base class are again not accessible in the derived class.

Given table will summarize the above three modes of inheritance and show the Base class member access specifier when derived in all three modes of inheritance in C++:

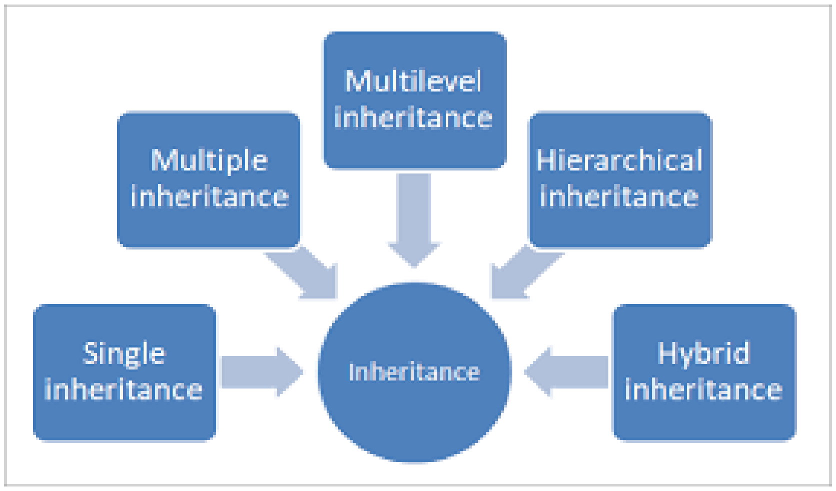


Base class visibility	**Derived class visibility**		
	Public	**Private**	**Protected**
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

➢ **Types of inheritance:**

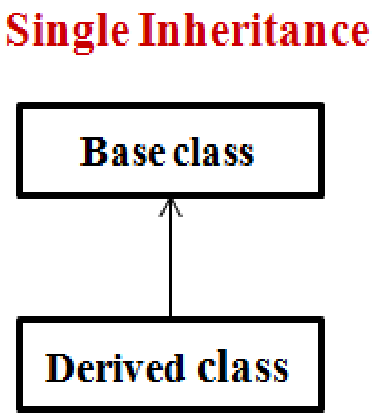
There are five types of inheritance:

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hybrid Inheritance
- Hierarchical Inheritance



- **Single Inheritance:**

When the derived class inherits only one base class, it is known as Single Inheritance.



In the above diagram, there is a Base class, and a derived class. Here the child class inherits only one parent class.

Syntax:

```
class subclass_name : access_mode base_class
{
  // body of subclass
};

OR

class A
{
... .. ...
};

class B: public A
{
... .. ...
};
```

Example of Single Inheritance:

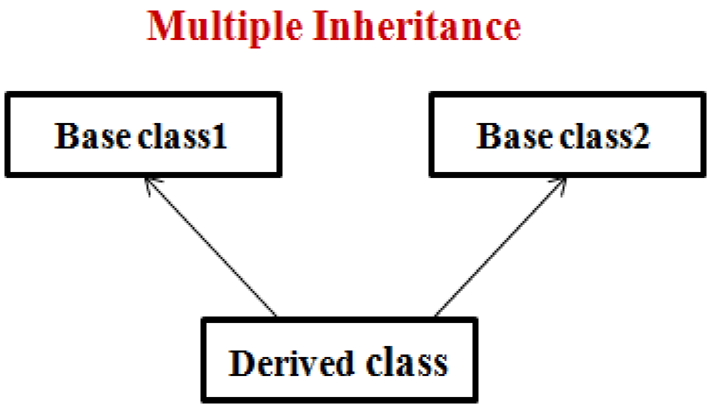
```
#include<iostream>
using namespace std;

// base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle\n";
    }
};

// sub class derived from a single base classes
class Car : public Vehicle {

};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Output:

```
This is a Vehicle
```

- **Multiple Inheritance:**

Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e one subclass is inherited from more than one base class.



Syntax:

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
  // body of subclass
};
class B
{
... .. ...
};
class C
{
... .. ...
};
class A: public B, public C
```

```
{
... ... ...
};
```

Example:

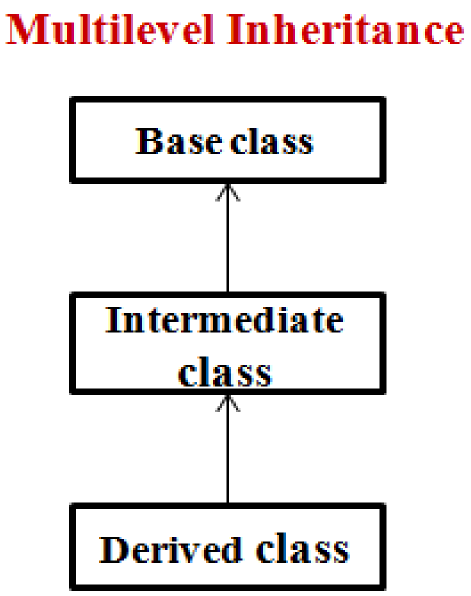
```
#include <iostream>
using namespace std;

// first base class
class Vehicle {
public:
   Vehicle() { cout << "This is a Vehicle\n"; }
};

// second base class
class FourWheeler {
public:
   FourWheeler()
   {
      cout << "This is a 4 wheeler Vehicle\n";
   }
};

// sub class derived from two base classes
class Car : public Vehicle, public FourWheeler {
};
// main function
int main()
{
   // Creating object of sub class will
   // invoke the constructor of base classes.
   Car obj;
   return 0;
}
```

Output :

```
This is a Vehicle
This is a 4 wheeler Vehicle
```

- **Multilevel Inheritance:**

In this type of inheritance, a derived class is created from another derived class.



Syntax:-

```
class C
{
... .. ...
};
class B:public C
{
... .. ...
};
class A: public B
{
... ... ...
};
```

Example:

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

// base class
class Vehicle {
```

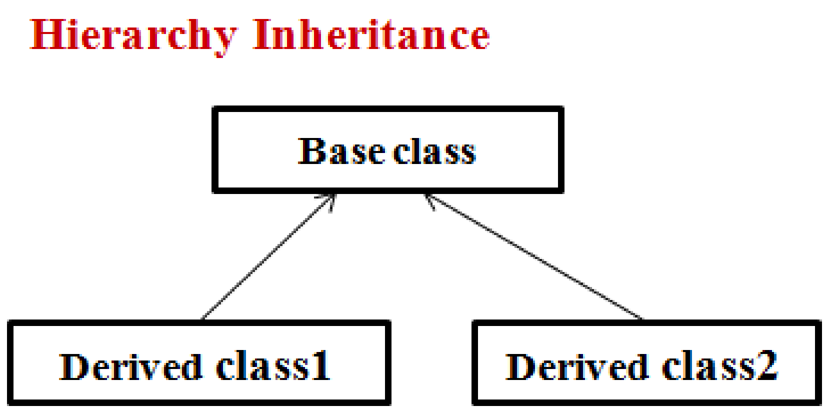
```
public:
   Vehicle() { cout << "This is a Vehicle\n"; }
};

// first sub_class derived from class vehicle
class fourWheeler : public Vehicle {
public:
   fourWheeler()
   {
      cout << "Objects with 4 wheels are vehicles\n";
   }
};
// sub class derived from the derived base class fourWheeler
class Car : public fourWheeler {
public:
   Car() { cout << "Car has 4 Wheels\n"; }
};
// main function
int main()
{
   // Creating object of sub class will
   // invoke the constructor of base classes.
   Car obj;
   return 0;
}
```

Output:

```
This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 Wheels
```

- **Hierarchical Inheritance:**

In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.



Syntax:-

```
class A
{
   // body of the class A.
}
class B : public A
{
   // body of class B.
}
class C : public A
{
   // body of class C.
}
class D : public A
{
   // body of class D.
}
```

Example:

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

// base class
class Vehicle {
public:
   Vehicle() { cout << "This is a Vehicle\n"; }
};
```

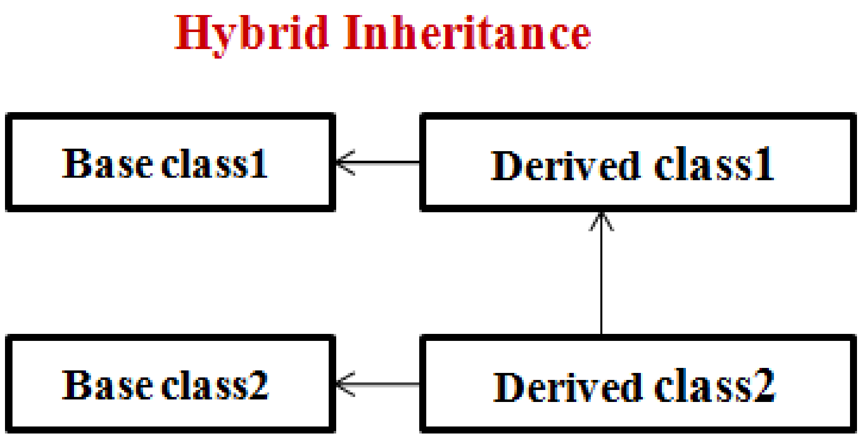
```
// first sub class
class Car : public Vehicle {
};

// second sub class
class Bus : public Vehicle {
};

// main function
int main()
{
   // Creating object of sub class will
   // invoke the constructor of base class.
   Car obj1;
   Bus obj2;
   return 0;
}
```

Output:

```
This is a Vehicle
This is a Vehicle
```

- **Hybrid (Virtual) Inheritance:**

Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.



Example:

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

// base class
class Vehicle {
public:
   Vehicle() { cout << "This is a Vehicle\n"; }
};

// base class
class Fare {
public:
   Fare() { cout << "Fare of Vehicle\n"; }
};

// first sub class
class Car : public Vehicle {
};

// second sub class
class Bus : public Vehicle, public Fare {
};
// main function
int main()
{
   // Creating object of sub class will
   // invoke the constructor of base class.
   Bus obj2;
   return 0;
}
```

Output:

```
This is a Vehicle
Fare of Vehicle
```

- **Virtual function**

  - A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
  - It is used to tell the compiler to perform dynamic linkage or late binding on the function.
  - There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
  - A 'virtual' is a keyword preceding the normal declaration of a function.
  - When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

By default, C++ matches a function call with the correct function definition at compile time. This is called static binding. You can specify that the compiler match a function call with the correct function definition at run time; this is called dynamic binding. You declare a function with the keyword virtual if you want the compiler to use dynamic binding for that specific function.

If we have created a virtual function in the base class and it is being overridden in the derived class then we don't need virtual keyword in the derived class, functions are automatically considered as virtual functions in the derived class.

- **Rules of Virtual Function:**

  - Virtual functions must be members of some class.
  - Virtual functions cannot be static members.
  - They are accessed through object pointers.
  - They can be a friend of another class.
  - A virtual function must be defined in the base class, even though it is not used.
  - The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
  - We cannot have a virtual constructor, but we can have a virtual destructor
  - Consider the situation when we don't use the virtual keyword.

```
// concept of Virtual Functions
#include<iostream>
using namespace std;

class base {
```

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

    void show()
    {
        cout << "show base class\n";
    }
};

class derived : public base {
public:
    void print()
    {
        cout << "print derived class\n";
    }
void show()
    {
        cout << "show derived class\n";
    }
};

int main()
{
    base *bptr;
    derived d;
    bptr = &d;

    // Virtual function, binded at runtime
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();

    return 0;
}
```

Output:

```
print derived class
show base class
```

Example:

```
// working of Virtual Functions
#include<iostream>
using namespace std;

class base {
public:
   void fun_1() { cout << "base-1\n"; }
   virtual void fun_2() { cout << "base-2\n"; }
   virtual void fun_3() { cout << "base-3\n"; }
   virtual void fun_4() { cout << "base-4\n"; }
};

class derived : public base {
public:
   void fun_1() { cout << "derived-1\n"; }
   void fun_2() { cout << "derived-2\n"; }
   void fun_4(int x) { cout << "derived-4\n"; }
};
int main()
{
   base *p;
   derived obj1;
   p = &obj1;

   // Early binding because fun1() is non-virtual
   // in base
   p->fun_1();

   // Late binding (RTP)
   p->fun_2();

   // Late binding (RTP)
   p->fun_3();
```

```
    // Late binding (RTP)
    p->fun_4();

    // Early binding but this function call is
    // illegal (produces error) because pointer
    // is of base type and function is of
    // derived class
    // p->fun_4(5);
        return 0;
        }
```

Output:

```
base-1
derived-2
base-3
base-4
```

Initially, we create a pointer of type base class and initialize it with the address of the derived class object. When we create an object of the derived class, the compiler creates a pointer as a data member of the class containing the address of VTABLE of the derived class.

Similar concept of Late and Early Binding is used as in above example. For fun_1() function call, base class version of function is called, fun_2() is overridden in derived class so derived class version is called, fun_3() is not overridden in derived class and is virtual function so base class version is called, similarly fun_4() is not overridden so base class version is called.

fun_4(int) in derived class is different from virtual function fun_4() in base class as prototypes of both the functions are different.

- **Limitations of Virtual Functions:**

- **Slower:** The function call takes slightly longer due to the virtual mechanism and makes it more difficult for the compiler to optimize because it does not know exactly which function is going to be called at compile time.

- **Difficult to Debug:** In a complex system, virtual functions can make it a little more difficult to figure out where a function is being called from.

- **Pure Virtual Function**

- ➢ A virtual function is not used for performing any task. It only serves as a placeholder.
- ➢ When the function has no definition, such function is known as "do-nothing" function.
- ➢ The "do-nothing" function is known as a pure virtual function. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- ➢ A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- ➢ The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

Pure virtual function can be defined as:

virtual void display() = 0;

Example:

```
#include<iostream>
using namespace std;
class B {
  public:
    virtual void s() = 0; // Pure Virtual Function
};

class D:public B {
  public:
    void s() {
      cout << "Virtual Function in Derived class\n";
    }
};

int main() {
  B *b;
  D dobj;
  b = &dobj;
  b->s();
}
```

Output:

```
Virtual Function in Derived class
```

- **Order of Constructors and Destructors in simple inheritance:**

In simple inheritance, the constructors of the base classes and constructors of the derived class are automatically executed when an object of the derived class is created. The constructors of the base classes are executed first and then the constructor of the derived class is executed. The constructors of base classes are executed in the same order in which the base classes are specified in derived class(i.e. from left to right).

Similarly, the destructors are executed in reverse order, i.e. derived class constructor is executed first and then constructors of the base classes are executed.

Example:

```
#include <iostream>
using namespace std;
class Base
{
public:
   Base()
   {
      cout<<"Constructor Base \n";
   }
   ~Base()
   {
      cout<<"Destructor Base \n";
   }

};

class Derived: public Base
{
public:
   Derived()
   {
      cout<<"Constructor Derived \n";
   }

   ~Derived()
   {
      cout<<"Destructor Derived \n";
```

```
    }
};
int main(void)
{
    Derived d;
    return 0;
}
```

Output:

```
Constructor Base
Constructor Derived
Destructor Derived
Destructor Base
```

- **Order of Constructor and destructor call for multiple inheritance**

During multiple inheritance, the constructor of base class will be called first.

The order of constructor called will be in the order of constructor defined.

For example if you have defined like this “class Derived: public A, public B”, then Constructor of class A will be called, then constructor of class B will be called.

Example:

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

class A
{
public:
    A()
    {
        cout<<"Constructor A \n";
    }
    ~A()
    {
        cout<<"Destructor A \n";
    }

};
```

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

};
class Derived: public A, public B
{
public:
   Derived()
   {
      cout<<"Constructor Derived \n";
   }

   ~Derived()
   {
      cout<<"Destructor Derived \n";
   }
};
int main(void)
{
   Derived d;
   return 0;
```

Output:

```
Constructor A
Constructor B
Constructor Derived
Destructor Derived
Destructor B
Destructor A
```

- **Polymorphism**

Polymorphism word is the combination of "poly," which means many + "morphs," which means forms, which together means many forms. Polymorphism in C++ is when the behavior of the same object or function is different in different contexts. Eg., the + operator in C++ is one of the best examples of polymorphism. The + operator is used to add two integers as follows :

```
#include <bits/stdc++.h>
using namespace std;

int main() {
   int a = 10;
   int b = 32;

   cout << "Value of a + b is: " << a + b;
         return 0;
}
```

Output: Value of a + b is: 42

In the above example, + is used to add a = 10a=10 and b = 32b=32, the output Value of a + ba+b is : 42.

However, + is also used to concatenate two string operands. The + operator is used to concatenate two strings as follows :

```
#include <bits/stdc++.h>
using namespace std;

int main() {
   string a = "poly";
   string b = "morphism";

   cout << "Value of a + b is: " << a + b;
         return 0;
}
```

Output: Value of a + b is: polymorphism
In the above example, + is used to concatenate two strings aa = poly and bb = morphism, therefore the output Value of a+ba+b is polymorphism.

- **Types of Polymorphism**

Following are the two types of polymorphism in C++:

1. Compile Time Polymorphism
2. Runtime Polymorphism

1. **Compile-Time Polymorphism:** Compile-time polymorphism is done by overloading an operator or function. It is also known as "static" or "early binding".

   - ***Why is it called compile-time polymorphism?***
     Overloaded functions are called by comparing the data types and number of parameters. This type of information is available to the compiler at the compile time. Thus, the suitable function to be called will be chosen by the C++ compiler at compilation time.

   There are the following types of compile-time polymorphism in C++:

   - Function overloading
   - Operator overloading

- **C++ Function Overloading:**

In C++, we can use two functions having the same name if they have different parameters (either types or number of arguments).

And, depending upon the number/type of arguments, different functions are called. For example,

```
// C++ program to overload sum() function

#include <iostream>
using namespace std;

// Function with 2 int parameters
int sum(int num1, int num2) {
   return num1 + num2;
}

// Function with 2 double parameters
double sum(double num1, double num2) {
   return num1 + num2;
}
```

```
// Function with 3 int parameters
int sum(int num1, int num2, int num3) {
   return num1 + num2 + num3;
}
int main() {
   // Call function with 2 int parameters
   cout << "Sum 1 = " << sum(5, 6) << endl;

   // Call function with 2 double parameters
   cout << "Sum 2 = " << sum(5.5, 6.6) << endl;

   // Call function with 3 int parameters
   cout << "Sum 3 = " << sum(5, 6, 7) << endl;

   return 0;
}
```

Output:

```
Sum 1 = 11
Sum 2 = 12.1
Sum 3 = 18
```

Here, we have created 3 different sum() functions with different parameters (number/type of parameters). And, based on the arguments passed during a function call, a particular sum() is called.

It's a **compile-time polymorphism** because the compiler knows which function to execute before the program is compiled.

- **C++ Operator overloading:**

In C++, we can overload an operator as long as we are operating on user-defined types like objects or structures. We cannot use operator overloading for basic types such as int, double, etc. Operator overloading is basically function overloading, where different operator functions have the same symbol but different operands.

And, depending on the operands, different operator functions are executed. For example,

```
// C++ program to 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 + 1;
   }

   void display() {
      cout << "Count: " << value << endl;
   }
};
int main() {
   Count count1;

   // Call the "void operator ++()" function
   ++count1;

   count1.display();
   return 0;
}
```

Output:

```
Count: 6
```

Here, we have overloaded the ++ operator, which operates on objects of Count class (object count1 in this case).
We have used this overloaded operator to directly increment the value variable of count1 object by 1.

This is also a **compile-time polymorphism.**

2. **Runtime Polymorphism:** Runtime polymorphism occurs when functions are resolved at runtime rather than compile time when a call to an overridden method is resolved dynamically at runtime rather than compile time. It's also known as late binding or dynamic binding.
   Runtime polymorphism is achieved using a combination of **function overriding** and **virtual functions**.

- **C++ Function Overriding:**

In C++ inheritance, we can have the same function in the base class as well as its derived classes.

When we call the function using an object of the derived class, the function of the derived class is executed instead of the one in the base class.

So, different functions are executed depending on the object calling the function.

This is known as **function overriding in C++.** For example,

```
// C++ program to demonstrate function overriding

#include <iostream>
using namespace std;

class Base {
  public:
   virtual void print() {
       cout << "Base Function" << endl;
   }
};

class Derived : public Base {
  public:
   void print() {
       cout << "Derived Function" << endl;
   }
};
int main() {
   Derived derived1;
```

```
    // Call print() function of Derived class
    derived1.print();

    return 0;
}
```

Output:

```
Derived Function
```

Here, we have used a print() function in the Base class and the same function in the Derived class
When we call print() using the Derived object derived1, it overrides the print() function of Base by executing the print() function of the Derived class.

It's a **runtime polymorphism** because the function call is not resolved by the compiler, but it is resolved in the runtime instead.

- **C++ Virtual Function:**

In C++, we may not be able to override functions if we use a pointer of the base class to point to an object of the derived class.
Using virtual functions in the base class ensures that the function can be overridden in these cases.
Thus, virtual functions actually fall under **function overriding**. For example,

```
// C++ program to demonstrate the use of virtual functions

#include <iostream>
using namespace std;

class Base {
  public:
   virtual void print() {
       cout << "Base Function" << endl;
   }
};

class Derived : public Base {
  public:
   void print() {
       cout << "Derived Function" << endl;
   }
};
int main() {
   Derived derived1;

   // pointer of Base type that points to derived1
   Base* base1 = &derived1;

   // calls member function of Derived class
   base1->print();

   return 0;
}
```

Output:

```
Derived Function
```

Here, we have used a virtual function print() in the Base class to ensure that it is overridden by the function in the Derived class.
Virtual functions are **runtime polymorphism.**

- **Unary Operator Overloading**

The unary operators are one such extensively used operator. Unary Operators work on the operations that are carried out on just one operand, both in mathematics and in programming languages. Unary operators do not utilize two operands to calculate the result as binary operators do.
A single operand/variable is used with the unary operator to determine the new value of that variable. Unary operators are used with the operand is either the prefix or postfix position. Unary operators come in a variety of forms, and they all have right-to-left associativity and equal precedence.
These are some of examples, unary operators:

- Increment operator (++),
- Decrement operator (- -),
- Unary minus operator (-),
- Logical not operator (!),

- Address of (&), etc.

Example:

Overloading Unary Operators for User-Defined Classes:

```
#include <iostream>
using namespace std;
class Complex
{
private:
   int real, img;

public:
   Complex()
   {
      real = 0;
      img = 0;
   }

   Complex(int r, int i)
   {
      real = r;
      img = i;
   }
}
```

There are two ways for unary operation overloading in C++, i.e.

- By adding the operator function as a class member function.
- By using the global friend function created for the operator function.

Example: C++ program to overload unary minus (-) to negate the values.

```
/*C++ program for unary minus (-) operator overloading.*/

#include<iostream>
using namespace std;

class NUM
{
   private:
        int n;

    public:
        //function to get number
        void getNum(int x)
        {
            n=x;
        }
        //function to display number
        void dispNum(void)
        {
            cout << "value of n is: " << n;
        }
        //unary - operator overloading
        void operator - (void)
        {
            n=-n;
        }
};
int main()
{
    NUM num;
    num.getNum(10);
    -num;
    num.dispNum();
    cout << endl;
    return 0;

}
```

Output:

```
    value of n is: -10
```

- **Binary Operator Overloading**

An operator which contains two operands to perform a mathematical operation is called the Binary Operator Overloading. It is a polymorphic compile technique where a single operator can perform various functionalities by taking two operands from the programmer or user.

Example : Program to add two numbers using the binary operator overloading

```
/* use binary (+) operator to perform the addition of two numbers. */

#include <iostream>
using namespace std;
class Arith_num
{
   // declare data member or variable
   int x, y;
   public:
      // create a member function to take input
      void input()
      {
         cout << " Enter the first number: ";
         cin >> x;
      }
      void input2()
      {
         cout << " Enter the second number: ";
         cin >> y;
      }
  // overloading the binary '+' operator to add number
      Arith_num operator + (Arith_num &ob)
      {
         // create an object
         Arith_num A;
         // assign values to object
         A.x = x + ob.x;
         return (A);
      }
      // display the result of binary + operator
      void print()
      {
         cout << "The sum of two numbers is: " <<x;
      }
};
int main ()
{
  Arith_num x1, y1, res; // here we create object of the class Arith_num i.e x1 and y1
```

```
    // accepting the values
    x1.input();
    y1.input();
     // assign result of x1 and x2 to res
    res = x1 + y1;
    // call the print() function to display the results
    res.print();
    return 0;
}
```

Output:

```
        Enter the first number: 5
        Enter the second number: 6
        The sum of two numbers is: 11
```