### Overview of Inheritance

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

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

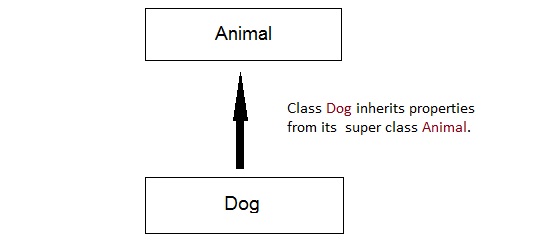
#### Basic Syntax of Inheritance

class Subclass\_name : access\_mode Superclass\_name

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



class Animal

{ public:

int legs = 4;

};

class Dog : public Animal

{ public:

int tail = 1;

};

int main()

{

Dog d;

cout << d.legs;

cout << d.tail;

}

Output : 4 1

### Inheritance Visibility Mode

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

#### 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

#### 3) Protected Inheritance

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

class subclass : **protected** Superclass

#### Table showing all the Visibility Modes

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

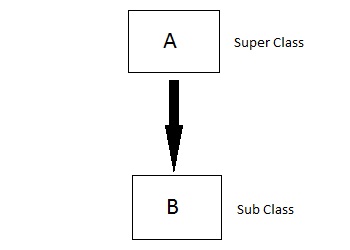
### Types of Inheritance

In C++, 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)

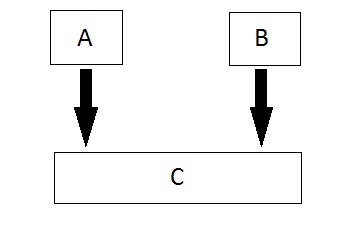
#### Single Inheritance

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



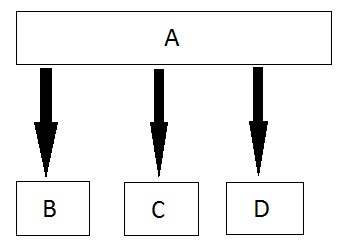
#### Multiple Inheritance

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



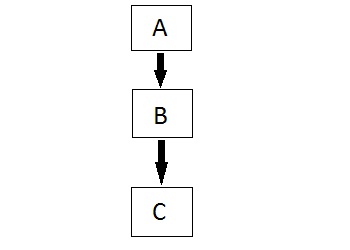
#### Hierarchical Inheritance

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



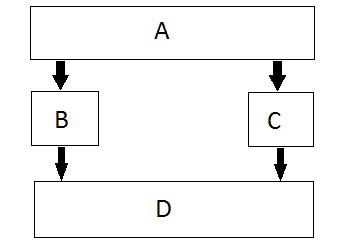
#### Multilevel Inheritance

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

Hybrid Inheritance is combination of Hierarchical and Mutilevel Inheritance.



### Order of Constructor Call

Base class constructors are always called in the derived class constructors. Whenever you create derived class object, first the base class default constructor is executed and then the derived class's constructor finishes execution.

#### Points to Remember

1. Whether derived class's default constructor is called or parameterised is called, base class's default constructor is always called inside them.
2. To call base class's parameterised constructor inside derived class's parameterised constructo, we must mention it explicitly while declaring derived class's parameterized constructor.

#### Base class Default Constructor in Derived class Constructors

class Base

{ int x;

public:

Base() { cout << "Base default constructor"; }

};

class Derived : public Base

{ int y;

public:

Derived() { cout << "Derived default constructor"; }

Derived(int i) { cout << "Derived parameterized constructor"; }

};

int main()

{

Base b;

Derived d1;

Derived d2(10);

}

You will see in the above example that with both the object creation of the Derived class, Base class's default constructor is called.

#### Base class Parameterized Constructor in Derived class Constructor

We can explicitly mention to call the Base class's parameterized constructor when Derived class's parameterized constructor is called.

class Base

{ int x;

public:

Base(int i)

{ x = i;

cout << "Base Parameterized Constructor";

}

};

class Derived : public Base

{ int y;

public:

**Derived(int j) : Base(j)**

{ y = j;

cout << "Derived Parameterized Constructor";

}

};

int main()

{

Derived d(10) ;

cout << d.x ; // Output will be 10

cout << d.y ; // Output will be 10

}

#### Why is Base class Constructor called inside Derived class ?

Constructors have a special job of initializing the object properly. A Derived class constructor has access only to its own class members, but a Derived class object also have inherited property of Base class, and only base class constructor can properly initialize base class members. Hence all the constructors are called, else object wouldn't be constructed properly.

#### Constructor call in Multiple Inheritance

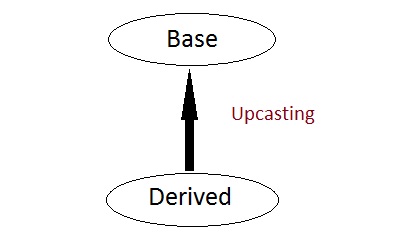
Its almost the same, all the Base class's constructors are called inside derived class's constructor, in the same order in which they are inherited.

class A : public B, public C ;

In this case, first class B constructor will be executed, then class C constructor and then class A constructor.

### 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;

}

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.
* Also, assignment operator = is never inherited. It can be overloaded but can't be inherited by sub class.

#### Inheritance and Static Functions

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 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 C:public A {};

class D:public B, public C {};

int main()

{

D obj;

obj.show();

}

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 C : virtual public A {};

class D : public B, public C {};

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.