Classes and objects

```
class Student {

string name;
int id;
float gpa;

public:
Student();
Student(string, int, float);
void inputStudent();
void outputStudent();
float getGpa();
void setGpa();
};
```

Save the class as Student.h

Create Student.cpp

```
#include "Student.h"
```

resolution operator

```
Student :: Student()

{
name = "";
id = 0;
gpa = 0.0;
}

Student :: Student(string name, int i, float g) {
this.name = name;
id = I;
gpa = g;
}

Student :: inputStudent() {
cin >> name;
cin >> id;
cin >> gpa;
}

Student:: outputStudent() {
cout << name << endl;
cout << id << endl;
cout << gpa << endl;
}


float Student :: getGpa() {
return gpa;
}
```

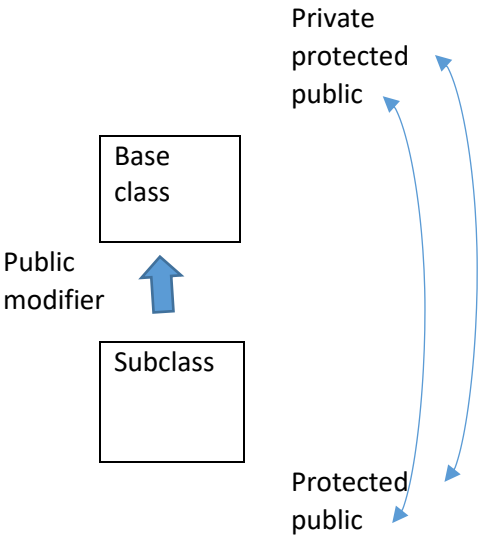
```
void Student :: stGpa(float f) {
gpa = f;
}
```

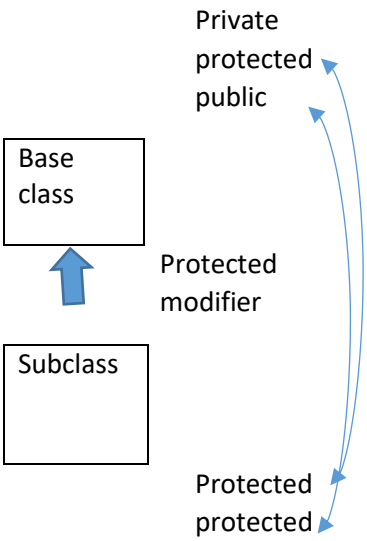
Implement the main function:

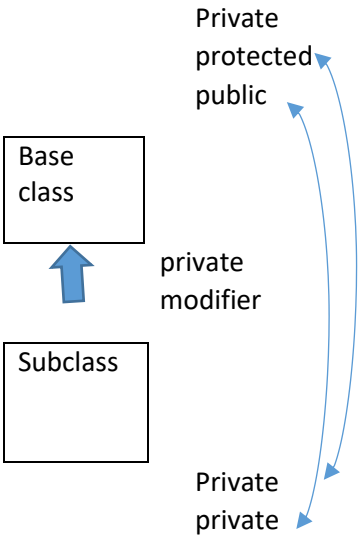
```
int main () {

Student stud1; // call Student()
Student stud2("John, 123, 3.5); // call Student()
stud1.inputStudent();
stud1.outputStudent();
stud2.setGpa(3.7);
cout << stud2.getGpa();
}
```

Inheritance

```
Derived class:   modifier      Base class
{                (public
                 protected
                 private)
}
```







Single inheritance

```
class Shape {
   protected:
            int x;
            int y;
  public:
            void setxy(int x1, int y1) {
            x = x1;
            y = y1;
            }
};
```

```
class Rectangle : public Shape {
float w;
float h;
  public:
        float area() {
        return w*h;
        }

        Rectangle(float w1, float h1) : Shape() {
        w = w1;
        h = h1;
        }

        void displayRec() {
        cout << “x = “ << x << endl;
        cout << “y = “ << y << endl;
        cout << “ Area = “ << area() << endl;
        }
};
```

In main:

```
Rectangle r1 (10.0, 20.0);
r1.setxt(3,5);
r1.displayRec(); // x= 3, y = 5, area = 200
```

Initialize data numbers in a constructor

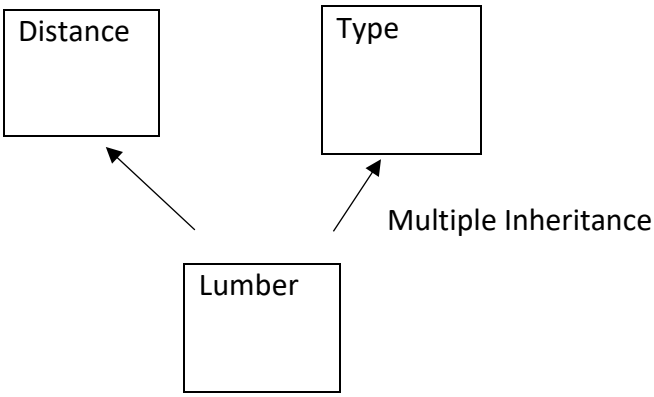
```
class Distance {

 int feet;
 float inches;
public:
        Distance() {
          feet = 0;
          inches = 0.0;
        }
OR:

        Distance() : float (0), inches(0.0)
        {
        }


        Distance (int f, float i)
        {
           feet = f;
           inches = I;
        }
        OR:
```

```
Distance(int f, float i) : feet(f), inches(i)
{
}
```

```
Distance (int f, float i)
{
      feet = f;
       inches = I;
}
```

Constructors in multiple inheritance

```
Class Type {

  string dimension;
  string grade;
  public:
    Type(): dimension("N/A") {  }
    Type(string d, string g): dimension(d), grade(y) {   }
```




```
class Lumber: public  Distance, public Type
   private:
      int quantity;
      float price;
  public:
      Lumber(): Distance(), Type() {
         quantity = 0;
         price = 0.0;
      }
```

OR:

```
      Lumber(): Distance(), Type(), quantity(0), price(0.0)  {
      }
```

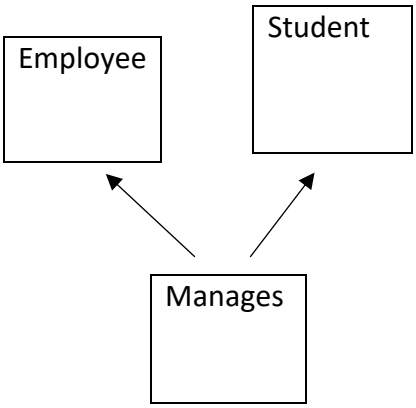
In main:

```
Lumber l1();
Lumber(int f, float i, string d, string g, int q, float p): Distance(f, i), Type(d, g), quantity(0), price(0.0)
```

Distance Type Lumber

```
{   }
```

Member functions in multiple inheritance




```
class Student
{
  private:
    string school,
    string degree;
  public:
     void getData() {
       cin >> school;
       cin >> degree;
     }
     void putData() {
       cout << school << endl;
       cout << degree << endl;
     }
};
```

```
class Employee {
  string name;
  int id;
  public:
     void getData() {
       cin >> name;
       cin >> id;
```

```
    }
    void putData(): const { // Cannot change any values inside the function.
      cout << name << endl;
      cout << id << endl;
    }

};

class Manager: private Employer, private Student {


  private:
    string title;
    float dues;
  public:
    void getData() {
      Employee:: getData();
      Student:: getData();
      cin >> title;
      cin >> dues;
    }
    void putData() {
      Employee:: putData();
      Student:: putData();
      cout << title << endl;
    }
};
```

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.

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

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

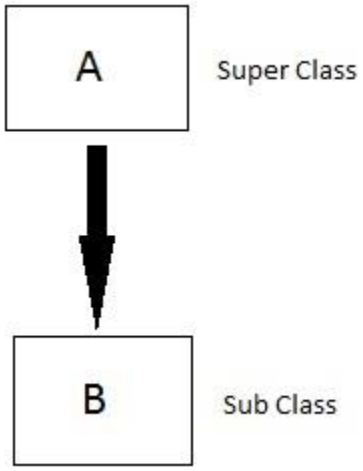
In C++, we have 5 different types of Inheritance. Namely,

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance

4. Multilevel Inheritance
5. Hybrid Inheritance (also known as Virtual Inheritance)

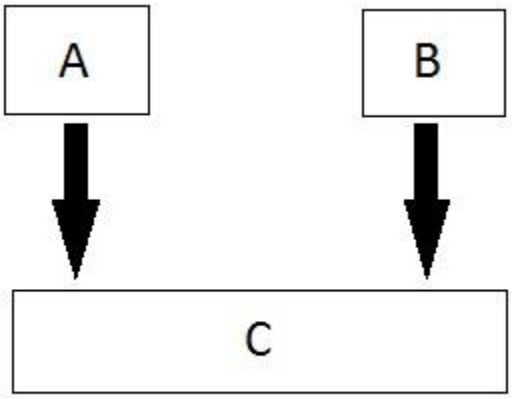
Single Inheritance

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



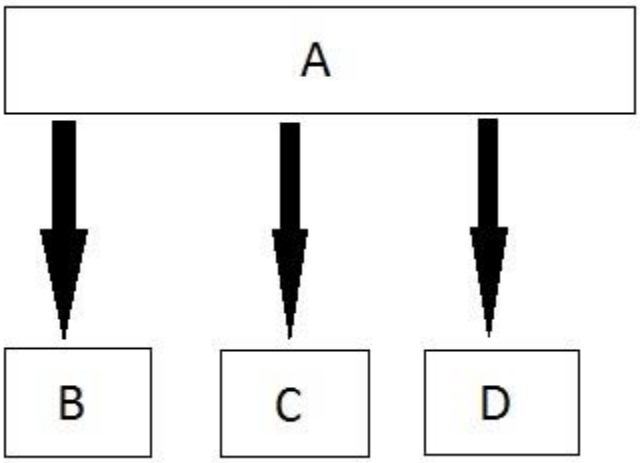
Multiple Inheritance

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



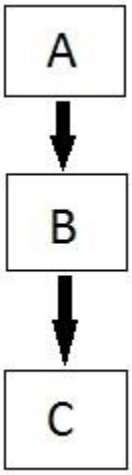
Hierarchical Inheritance

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



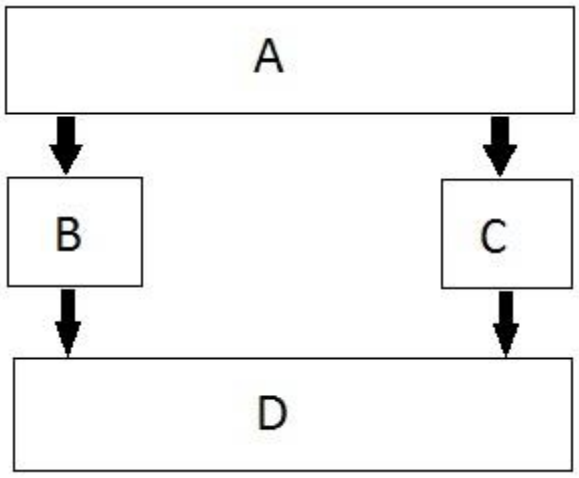
Multilevel Inheritance

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\n";
  }
};

class Derived : public Base
{ int y;
  public:
  Derived(int j) : Base(j)
  { y = j;
    cout << "Derived Parameterized Constructor\n";
  }
};

int main()
{
 Derived d(10) ;
 getch();
}
```

Output:

```
Base Parameterized Constructor
Derived Parameterized Constructor
```

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

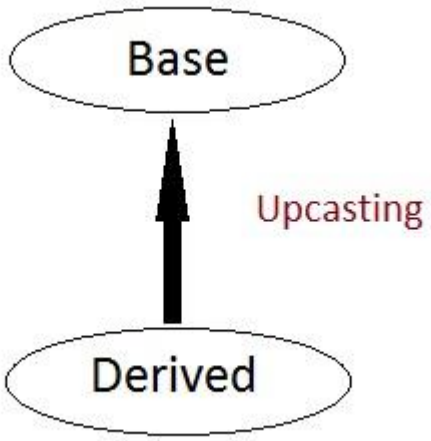
Its almost the same, all the Base class's constructors are called inside derived class's constructor, in the same order in which they are inherited.

```
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.

Polymorphism

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.

Function Overriding

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

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

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

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

int main()
{
 Base b;      //Base class object
```

```
 Derived d;    //Derived class object
 b.show();    //Early Binding Ocuurs
 d.show();
}
```

Output : 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";
 }
};
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 Occurs
}
```

Output : 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

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 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

```
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
}
```

Output : 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

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";
 }
};
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
}
```

```
Output : 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.

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.

Abstract Class

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

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;
```

Example of Abstract Class

```
class Base         //Abstract base class
{
 public:
 virtual void show() = 0;          //Pure Virtual Function
};

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

int main()
{
 Base obj;      //Compile Time Error
 Base *b;
 Derived d;
```

```
 b = &d;
 b->show();
}
```

Output : 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 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.

```
class Base         //Abstract base class
{
 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"; }
};

int main()
```

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

Output : Implementation of Virtual Function in Derived class

Virtual Destructors

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

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

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

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

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

Output : 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

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

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

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

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

Output :

```
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

- 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
};

Base::~Base() { cout << "Base Destructor"; } //Definition of Pure Virtual Destructor
```

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