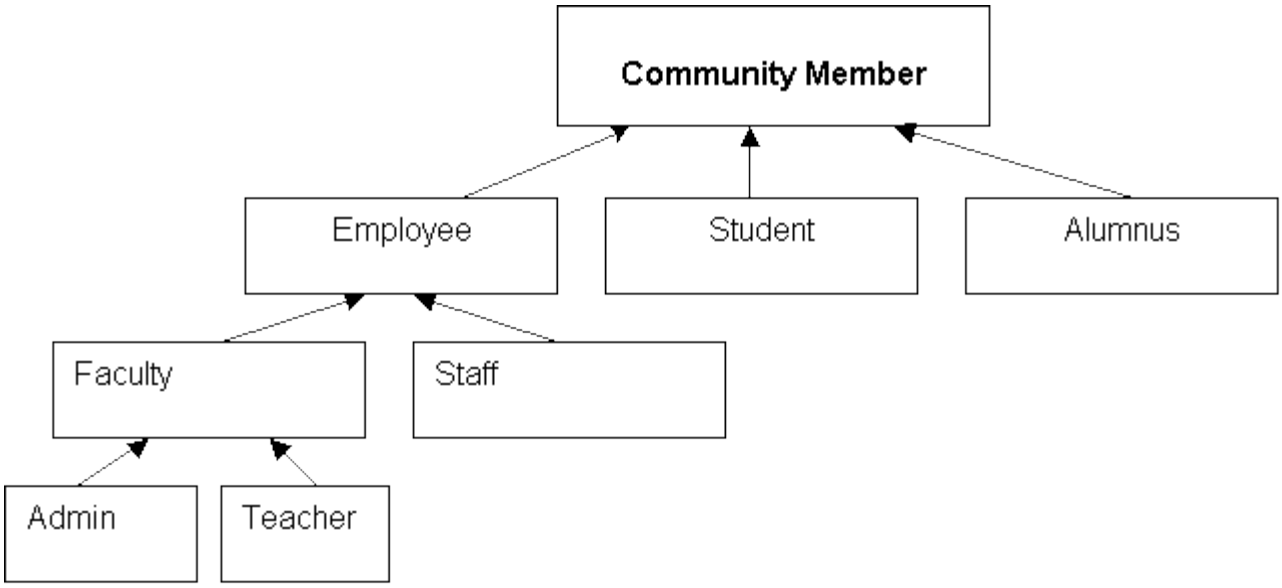
UNIT-3 PART-1

INHERITANCE

Defining class hierarchy:- The class hierarchy defines the inheritance relationship between the classes.

Class Hierarchy Example



- The arrows in the hierarchy represent is-a relationships. There are a number of direct superclass relationships and indirect superclass relationships.

Inheritance:-

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

Advantage of Inheritance

Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

C++ supports five types of inheritance:

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance

Derived Classes

A Derived class is defined as the class derived from the base class.

The Syntax of Derived class:

```
class derived_class_name :: visibility-mode base_class_name
{
   // body of the derived class.
}
```

Where,

derived_class_name: It is the name of the derived class.

visibility mode: The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.

base_class_name: It is the name of the base class.

- When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.

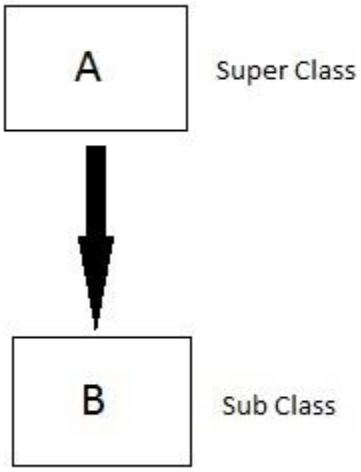
- When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the base class.

Note:

- In C++, the default mode of visibility is private.
- The private members of the base class are never inherited.

C++ Single Inheritance

Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.



Where 'A' is the base class, and 'B' is the derived class.

C++ Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

```
1. #include <iostream>
2. using namespace std;
```

```
3.  class Account {
4.    public:
5.    float salary = 60000;
6.  };
7.    class Programmer: public Account {
8.    public:
9.    float bonus = 5000;
10.  };
11.int main(void) {
12.     Programmer p1;
13.     cout<<"Salary: "<<p1.salary<<endl;
14.     cout<<"Bonus: "<<p1.bonus<<endl;
15.    return 0;
16.}
```

Output:

```
Salary: 60000
Bonus: 5000
```

In the above example, Employee is the **base** class and Programmer is the **derived** class.

C++ Single Level Inheritance Example: Inheriting Methods

Let's see another example of inheritance in C++ which inherits methods only.

```
1.  #include <iostream>
2.  using namespace std;
3.   class Animal {
4.    public:
5.   void eat() {
6.     cout<<"Eating..."<<endl;
7.   }
8.    };
9.    class Dog: public Animal
10.   {
```

```
11.       public:
12.     void bark(){
13.    cout<<"Barking...";
14.     }
15.   };
16.int main(void) {
17.    Dog d1;
18.    d1.eat();
19.    d1.bark();
20.    return 0;
21.}
```

Output:

```
Eating...
Barking...
```

Let's see a simple example.

```
1.  #include <iostream>
2.  using namespace std;
3.  class A
4.  {
5.      int a = 4;
6.      int b = 5;
7.      public:
8.      int mul()
9.      {
10.         int c = a*b;
11.         return c;
12.     }
13.};
14.
15.class B : private A
16.{
17.    public:
18.    void display()
19.    {
```

```
20.        int result = mul();
21.        std::cout <<"Multiplication of a and b is : "<<result<< std::endl;
22.    }
23.};
24.int main()
25.{
26.   B b;
27.   b.display();
28.
29.    return 0;
30.}
```

Output:

```
Multiplication of a and b is : 20
```

In the above example, class A is privately inherited. Therefore, the mul() function of class 'A' cannot be accessed by the object of class B. It can only be accessed by the member function of class B.

How to make a Private Member Inheritable

The private member is not inheritable. If we modify the visibility mode by making it public, but this takes away the advantage of data hiding.

C++ introduces a third visibility modifier, i.e., **protected**. The member which is declared as protected will be accessible to all the member functions within the class as well as the class immediately derived from it.

Visibility modes can be classified into three categories:

- **Public**: When the member is declared as public, it is accessible to all the functions of the program.
- **Private**: When the member is declared as private, it is accessible within the class only.
- **Protected**: When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

Visibility of Inherited Members

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

C++ Multilevel Inheritance

Multilevel inheritance is a process of deriving a class from another derived class.

C++ Multi Level Inheritance Example

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Let's see the example of multi level inheritance in C++.

```
1. #include <iostream>
2. using namespace std;
3.  class Animal {
4.    public:
5.  void eat() {
6.     cout<<"Eating..."<<endl;
7.  }
8.    };
9.    class Dog: public Animal
10.   {
```

```
11.       public:
12.     void bark(){
13.    cout<<"Barking..."<<endl;
14.     }
15.   };
16.   class BabyDog: public Dog
17.   {
18.       public:
19.     void weep() {
20.    cout<<"Weeping...";
21.     }
22.   };
23.int main(void) {
24.    BabyDog d1;
25.    d1.eat();
26.    d1.bark();
27.     d1.weep();
28.     return 0;
29.}
```

Output:

```
Eating...
Barking...
Weeping...
```

C++ Multiple Inheritance

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.

Syntax of the Derived class:

```
1. class D : visibility B-1, visibility B-2, ?
2. {
3.     // Body of the class;
```

```
4. }
```

Let's see a simple example of multiple inheritance.

```
1. #include <iostream>
2. using namespace std;
3. class A
4. {
5.     protected:
6.      int a;
7.     public:
8.     void get_a(int n)
9.     {
10.        a = n;
11.    }
12.};
13.
14.class B
15.{
16.    protected:
17.    int b;
18.    public:
19.    void get_b(int n)
20.    {
21.        b = n;
22.    }
23.};
24.class C : public A,public B
25.{
26.   public:
27.    void display()
28.    {
29.        std::cout << "The value of a is : " <<a<< std::endl;
30.        std::cout << "The value of b is : " <<b<< std::endl;
31.        cout<<"Addition of a and b is : "<<a+b;
32.    }
```

```
};
int main()
{
   C c;
   c.get_a(10);
   c.get_b(20);
   c.display();

    return 0;
}
```

Output:

```
The value of a is : 10
The value of b is : 20
Addition of a and b is : 30
```

In the above example, class 'C' inherits two base classes 'A' and 'B' in a public mode.

Ambiquity Resolution in Inheritance

Ambiguity can be occurred in using the multiple inheritance when a function with the same name occurs in more than one base class.

Let's understand this through an example:

```
#include <iostream>
using namespace std;
class A
{
    public:
    void display()
    {
        std::cout << "Class A" << std::endl;
    }
};
class B
{
```

```
13.    public:
14.    void display()
15.    {
16.        std::cout << "Class B" << std::endl;
17.    }
18.};
19.class C : public A, public B
20.{
21.    void view()
22.    {
23.        display();
24.    }
25.};
26.int main()
27.{
28.    C c;
29.    c.display();
30.    return 0;
31.}
```

Output:

```
error: reference to 'display' is ambiguous
        display();
```

- The above issue can be resolved by using the class resolution operator with the function. In the above example, the derived class code can be rewritten as:

```
1.  class C : public A, public B
2.  {
3.      void view()
4.      {
5.          A :: display();       // Calling the display() function of class A.
6.          B :: display();       // Calling the display() function of class B.
7.
8.      }
9.  };
```

An ambiguity can also occur in single inheritance.

Consider the following situation:

```
1. class A
2. {
3.    public:
4. void display()
5. {
6.    cout<<?Class A?;
7. }
8. } ;
9. class B
10.{
11.  public:
12. void display()
13.{
14. cout<<?Class B?;
15.}
16.} ;
```

In the above case, the function of the derived class overrides the method of the base class. Therefore, call to the display() function will simply call the function defined in the derived class. If we want to invoke the base class function, we can use the class resolution operator.

```
1. int main()
2. {
3.     B b;
4.    b.display();              // Calling the display() function of B class.
5.    b.B :: display();       // Calling the display() function defined in B class.
6. }
```

C++ Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.

Let's see a simple example:

```
#include <iostream>
using namespace std;
class A
{
  protected:
  int a;
   public:
   void get_a()
1.    {
2.      std::cout << "Enter the value of 'a' : " << std::endl;
3.      cin>>a;
4.    }
5. };
6.
7. class B : public A
8. {
9.    protected:
10.   int b;
11.   public:
12.   void get_b()
13.   {
14.      std::cout << "Enter the value of 'b' : " << std::endl;
15.     cin>>b;
16.   }
17.};
18.class C
19.{
20.   protected:
21.   int c;
22.   public:
23.   void get_c()
24.   {
25.      std::cout << "Enter the value of c is : " << std::endl;
26.      cin>>c;
```

```
27.    }
28.};
29.
30.class D : public B, public C
31.{
32.    protected:
33.    int d;
34.    public:
35.    void mul()
36.    {
37.        get_a();
38.        get_b();
39.        get_c();
40.        std::cout << "Multiplication of a,b,c is : " <<a*b*c<< std::endl;
41.    }
42.};
43.int main()
44.{
45.    D d;
46.    d.mul();
47.    return 0;
48.}
```

Output:

```
Enter the value of 'a' :
10
Enter the value of 'b' :
20
Enter the value of c is :
30
Multiplication of a,b,c is : 6000
```

C++ Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.

Syntax of Hierarchical inheritance:

```
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.
}
```

Let's see a simple example:

```
#include <iostream>
using namespace std;
class Shape                // Declaration of base class.
{
    public:
    int a;
    int b;
    void get_data(int n,int m)
    {
        a= n;
        b = m;
    }
};
class Rectangle : public Shape  // inheriting Shape class
{
    public:
```

```
17.    int rect_area()
18.    {
19.        int result = a*b;
20.        return result;
21.    }
22.};
23.class Triangle : public Shape    // inheriting Shape class
24.{
25.    public:
26.    int triangle_area()
27.    {
28.        float result = 0.5*a*b;
29.        return result;
30.    }
31.};
32.int main()
33.{
34.    Rectangle r;
35.    Triangle t;
36.    int length,breadth,base,height;
37.    std::cout << "Enter the length and breadth of a rectangle: " << std::endl;
38.    cin>>length>>breadth;
39.    r.get_data(length,breadth);
40.    int m = r.rect_area();
41.    std::cout << "Area of the rectangle is : " <<m<< std::endl;
42.    std::cout << "Enter the base and height of the triangle: " << std::endl;
43.    cin>>base>>height;
44.    t.get_data(base,height);
45.    float n = t.triangle_area();
46.    std::cout <<"Area of the triangle is : "  << n<<std::endl;
47.    return 0;
48.}
```

Output:

```
Enter the length and breadth of a rectangle:
23
```

```
20
Area of the rectangle is : 460
Enter the base and height of the triangle:
2
5
Area of the triangle is : 5
```

UNIT – 3 PART - 2

Virtual Functions and Polymorphism: Static and Dynamic binding, virtual functions, Dynamic binding through virtual functions, Virtual function call mechanism, Pure virtual functions, Abstract classes, Implications of polymorphic use of classes, Virtual destructors.

Binding:

Association of a ‘function definition’ to a ‘function call’ or an association of a ‘value’ to a ‘variable’, is called ‘binding’.

Static and Dynamic Binding

During compilation, every ‘function definition’ is given a memory address; as soon as function calling is done, control of program execution moves to that memory address and get the function code stored at that location executed, this is binding of ‘function call’ to ‘function definition’. Binding can be classified as ‘static binding’ and ‘dynamic binding’. If it’s already known before runtime, which function will be invoked or what value is allotted to a variable, then it is a ‘static binding’, and if it comes to know at the run time then it is called ‘dynamic binding’.

Basic differences between static and dynamic binding

BASIS FOR COMPARISON	STATIC BINDING	DYNAMIC BINDING
Event Occurrence	Events occur at compile time are "Static Binding".	Events occur at run time are "Dynamic Binding".
Information	All information needed to call a function is known at compile time.	All information need to call a function come to know at run time.
Advantage	Efficiency.	Flexibility.
Time	Fast execution.	Slow execution.
Alternate name	Early Binding.	Late Binding.
Example	overloaded function call, overloaded operators.	Virtual function in C++, overridden methods in java.

Explaination

1. Events that occur at compile time like, a function code is associated with a function call or assignment of value to a variable, are called **static/early binding**, and when these tasks are accomplished during runtime they are called **dynamic/late binding.**
2. **'Efficiency'** increases in static binding, as all the data is gathered before the execution. But in dynamic binding, the data is acquired at runtime so we can decide what value to assign a variable and which function to invoke at runtime this make execution 'flexible'.
3. **'Static binding'** make execution of a program **'faster'** as all the data needed to execute a program is known before execution. In **'dynamic binding'** data needed to execute a program is known to the compiler at the time of execution which takes the time to bind values to identifiers hence, it makes program execution **slower**.
4. Static binding is also called **early binding** because the function code is associated with function call during **compile time**, which is earlier than dynamic binding in which function code is associated with function call during **runtime** hence it is also called **late binding**.

Static Binding or Early Binding (compile-time time polymorphism)

As the name indicates, compiler (or linker) directly associate an address to the function call. It replaces the call with a machine language instruction that tells the mainframe to leap to the address of the function.**(or)**

When compiler acknowledges all the information required to call a function or all the values of the variables during compile time, it is called **"static binding"**. As all the required information are known before runtime, it increases the program efficiency and it also enhances the speed of execution of a program.

By default early binding happens in C++.

Implementation of static binding in C++ with example of overloading

```
#include<iostream>

using namespace std;

class overload{

int a, b;

public:

int load(int x){    // first load() function.

a=x;

return a;

}

int load(int x, int y){   //second load() function.

a=x;
```

```
b=y;
return a*b;
}
};
int main(){
overload O1;
cout<<O1.load(20)<<endl;  //This statement binds the calling of function to 'first'
                                load() function.
cout<<O1.load(20,40)<<endl;  //This statement binds the calling of function'second'
                                                load()function.
}
```

Output :

```
20
800
```

/* CPP Program to illustrate early binding. Any normal function call (without virtual) is binded early. */

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

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

class Derived: public Base
```

```
{
public:
        void show() { cout<<"In Derived \n"; }
};


int main(void)
{
        Base *bp = new Derived; /*The function call decided at compile time (compiler sees
                                                        type of pointer and calls base class function. */
        bp->show();


        return 0;
}
```

Output : In Base

Dynamic Binding or Late Binding

Late Binding or (Run time polymorphism) In this, the compiler adds code that identifies the kind of object at runtime then matches the call with the right function definition . This can be achieved by declaring a virtual function.

Dynamic binding can be associated with run time ‘polymorphism’ and ‘inheritance’ in OOP. Dynamic binding makes the execution of program flexible as it can be decided, what value should be assigned to the variable and which function should be called, at the time of program execution. But as this information is provided at run time it makes the execution slower as compared to static binding.

Implementation of dynamic binding using ‘virtual functions’ in C++.

```
#include<iostream>

using namespace std;

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

```
};
class Derived: public Base
{
public:
        void show() { cout<<"In Derived \n"; }
};
int main(void)
{
        Base *bp = new Derived;
        bp->show(); // RUN-TIME POLYMORPHISM
        return 0;
}
```

Virtual Functions in C++

- A virtual function a member function which is declared within a base class and is re-defined(Overriden) by a derived class.
- When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.
- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a virtual keyword in base class.
- The resolving of function call is done at Run-time.

Rules for Virtual Functions

1. Virtual functions cannot be static and also cannot be a friend function of another class.
2. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
3. The prototype of virtual functions should be same in base as well as derived class.
4. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
5. A class may have virtual destructor but it cannot have a virtual constructor.

6. Compile-time(early binding) VS run-time(late binding) behavior of Virtual Functions
7. 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.

Consider the following simple program showing run-time behavior of virtual functions.

```
// CPP program to illustrate concept of Virtual Functions
#include<iostream>
using namespace std;
class base
{
public:
        virtual void print ()
        { cout<<"print base class"<<endl; }
        void show ()
        { cout<<"show base class"<<endl; }
};
class derived:public base
{
public:
        void print ()
        { cout<<"print derived class"<<endl; }

        void show ()
        { cout<<"show derived class"<<endl; }
};
int main()
```

```
{
        base *bptr;
        derived d;
        bptr = &d;
        //virtual function, binded at runtime
        bptr->print();
        // Non-virtual function, binded at compile time
        bptr->show();
}
```

Output:

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

Pure virtual functions

A virtual function will become pure virtual function when you append "=0" at the end of declaration of virtual function. Pure virtual function doesn't have body or implementation. We must implement all pure virtual functions in derived class.

If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.

Pure virtual function is also known as abstract function.

A class with at least one pure virtual function or abstract function is called abstract class. We can't create an object of abstract class. Member functions of abstract class will be invoked by derived class object.

An abstract class can have constructors.

// Example: An abstract class with constructor

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

```
{
protected:
int x;
public:
virtual void fun() = 0;
Base(int i) { x = i; }
};


class Derived: public Base
{
        int y;
public:
        Derived(int i, int j):Base(i) { y = j; }
        void fun() { cout <<"x = "<< x <<", y = "<< y; }
};
int main(void)
{
        Derived d(4, 5);
        d.fun();
        return 0;
}
```

Output :x = 4, y = 5

Abstract class

Abstract class is used in situation, when we have partial set of implementation of methods in a class. For example, consider a class have four methods. Out of four methods, we have an

implementation of two methods and we need derived class to implement other two methods. In these kind of situations, we should use abstract class.

A virtual function will become pure virtual function when you append "=0" at the end of declaration of virtual function.

A class with at least one **pure virtual function** or **abstract function** is called abstract class.

Pure virtual function is also known as abstract function.

- We can't create an object of abstract class b'coz it has partial implementation of methods.
- Abstract function doesn't have body
- We must implement all abstract functions in derived class.

Example: Abstract Class and Pure Virtual Function

```
#include <iostream>

using namespace std;

// Abstract class

class Shape

{

   protected:

      float l;

   public:

      void getData()

      {

         cin >> l;

      }

      // virtual Function

      virtual float calculateArea() = 0;

};

class Square : public Shape

{
```

```
    public:
      float calculateArea()
      {   return l*l;  }
};
class Circle : public Shape
{
    public:
      float calculateArea()
      { return 3.14*l*l; }
};
int main()
{
    Square s;
    Circle c;
    cout <<"Enter length to calculate the area of a square: ";
    s.getData();
    cout<<"Area of square: "<< s.calculateArea();
    cout<<"\nEnter radius to calculate the area of a circle: ";
    c.getData();
    cout <<"Area of circle: "<< c.calculateArea();
    return 0;
}
```

Output

```
Enter length to calculate the area of a square: 4
Area of square: 16
```

Enter radius to calculate the area of a circle: 5

Area of circle: 78.5

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;
}
```

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 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;

}
```

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 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";
   }
};
```