UNIT III

OBJECT-ORIENTED PROGRAMMING CONCEPTS

UNITIII

OBJECT-ORIENTED PROGRAMMING CONCEPTS

- **Topics to be discussed,**

➢ Inheritance

➢ Constructors and Destructors in Derived Classes

➢ Polymorphism and Virtual Functions

UNITIII

OBJECT-ORIENTED PROGRAMMING CONCEPTS

- **Topics to be discussed,**

- **Inheritance**
- Constructors and Destructors in Derived Classes
- Polymorphism and Virtual Functions

Inheritance

- Inheritance is **one of the most important feature of Object Oriented Programming**.
- The capability of a class to derive properties and characteristics from another class is called **Inheritance**.
- It allows user to create a **new class (derived class)** from an **existing class(base 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.
- The idea of inheritance implements the **is a relationship**.
- For example, mammal IS-A animal, dog **IS-A** mammal hence dog IS-A animal as well and so on.

Inheritance – Cont'd

- **Sub Class:** The class which inherits properties of other class is called Child or Derived or Sub class. The derived class is the specialized class for the base class.
- **Super Class:** The class whose properties are inherited by other class is called the Parent or Base or Super class.
- **NOTE:- All members of a class except Private, are inherited.**
- A class can be derived from **more than one classes**, which means it can inherit data and functions from multiple base classes.

Inheritance – Cont'd

- **Syntax of Inheritance**

  class Derivedclass_name : access-specifier Baseclass_name

  {

  // body of subclass

  };

  - **Derivedclass_name** is the name of the derived class
  - **access-specifier** is the mode in which we want to inherit this sub class, i.e **public**, **protected**, or **private**.
  - **Base_class_name** is the name of the base class from which we want to inherit the sub class.
- If the access-specifier is not used, then it is **private** by default.
- **Note**: A derived class doesn't inherit ***access*** to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.

Inheritance – Cont'd

- **Modes of Inheritance:**
- There are 3 modes of inheritance.
  - **Public Mode**: If we derive a subclass from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in the derived class.
  - **Protected Mode**: If we derive a subclass from a Protected base class. Then both public members and protected members of the base class will become protected in the derived class.
  - **Private Mode**: If we derive a subclass from a Private base class. Then both public members and protected members of the base class will become Private in the derived class.
- **Note:** *The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed.*

Inheritance – Cont'd

- **Example:**

```
1. class Derived : private Base        //private derivation
        {           }
2. class Derived : public Base         //public derivation
        {           }
3. class Derived : protected Base  //protected derivation
        {           }
4. class Derived : Base          //private derivation by default
        {         }
```

Note:

- When a base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class and therefore, the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class.
- On the other hand, 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 derived class.

```
// C++ Implementation to show that a derived class
// doesn't inherit access to private data members.
// However, it does inherit a full parent object.
class A
{
    public:
        int x;

    protected:
        int y;

    private:
        int z;
};
class B : public A
{
    // x is public
    // y is protected
    // z is not accessible from B
};
class C : protected A
{
    // x is protected
    // y is protected
    // z is not accessible from C
};
class D : private A // 'private' is default for classes
{
    // x is private
    // y is private
    // z is not accessible from D
};
```

public derivation Example

```
#include <iostream>
using namespace std;
class Base
{
    int a;//Not Inheritable
    public:
        int b;//Inheritable
    void setAB(int x,int y)
    {
        a=x;b=y;
    }
    int getA()
    {
        return a;
    }
    void showA()
    {
        cout << "\na:"<<a<<endl;
    }
};
```

```
class Derived : public Base
{
    int c;

public:
    void mul()
    {
        c=b*getA();
    }

    void show()
    {

        cout <<"b: "<< b;
        cout << "\nc: "<< c;
    }
};
int main()
{
    Derived d;
    d.setAB(2,5);
    d.showA();
    d.mul();
    d.show();

    d.b=100;
    d.mul();
    d.showA();
    d.show();
    return 0;
}
```

Output:

```
a:2
b: 5
c: 10
a:2
b: 100
c: 200
```

private derivation Example

```
#include <iostream>
using namespace std;
class Base
{
    int a;//Not Inheritable
    public:
        int b;//Inheritable
    void setAB(int x,int y)
    {
        a=x;b=y;
    }
    int getA()
    {
        return a;
    }
    void showA()
    {
        cout << "\na:"<<a<<endl;
    }
};
class Derived : private Base
{
    int c;

public:
    void mul()
    {
        c=b*getA();
    }

    void show()
    {
        cout <<"\nb: "<< b;
        cout << "\nc: " << c;
    }
};
```

```
int main()
{
    Derived d;
    d.setAB(2,5);
    d.mul();
    d.showA();
    d.show();

    d.b=100;
    d.mul();
    d.show();
    return 0;
}
```

Output:

File	Line	Message
		=== Build file: "no target" in "no project" (compiler: unknown) ===
F:\2024\CS320...		In function 'int main()':
F:\2024\CS320...	41	error: 'void Base::setAB(int, int)' is inaccessible within this context
F:\2024\CS320...	8	note: declared here
F:\2024\CS320...	41	error: 'Base' is not an accessible base of 'Derived'
F:\2024\CS320...	43	error: 'void Base::showA()' is inaccessible within this context
F:\2024\CS320...	16	note: declared here
F:\2024\CS320...	43	error: 'Base' is not an accessible base of 'Derived'
F:\2024\CS320...	46	error: 'int Base::b' is inaccessible within this context
F:\2024\CS320...	7	note: declared here
		=== Build failed: 5 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===

Inheritance – Cont'd

- **Access Control and Inheritance:**

  class derived-class: access-specifier base-classA

  – derived class can access all the non-private members of its base class.

  – Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

- **A derived class can inherit all base class methods except**:
  - Constructors, destructors and copy constructors of the base class.
  - Overloaded operators of the base class.
  - The friend functions of the base class.

- Create a class **shape** with ***width*** and ***height*** as its data members and setWidth() and setHeight() as member functions which assigns the arguments received to width and height data members respectively. Create a class called Rectangle which inherits from Shape and has a member function getArea() which returns the area by finding the product of width and height. Create an object of Rectangle. Assign values to width and height and calculate area.

Inheritance - Example

```
#include <iostream>
using namespace std;
class Shape // Base class
{
protected:
  int width;
  int height;
public:
  void setWidth(int w)
  {
    width = w;
  }
  void setHeight(int h)
  {
    height = h;
  }
};
class Rectangle: public Shape// Derived class
 {
  public:
  int getArea()
  {
    return (width * height);
  }
};
```

```
int main()
{
  Rectangle Rect;
  Rect.setWidth(5);
  Rect.setHeight(7);
  cout << "Total area: " << Rect.getArea() << endl;
  return 0;
}
```

Output:

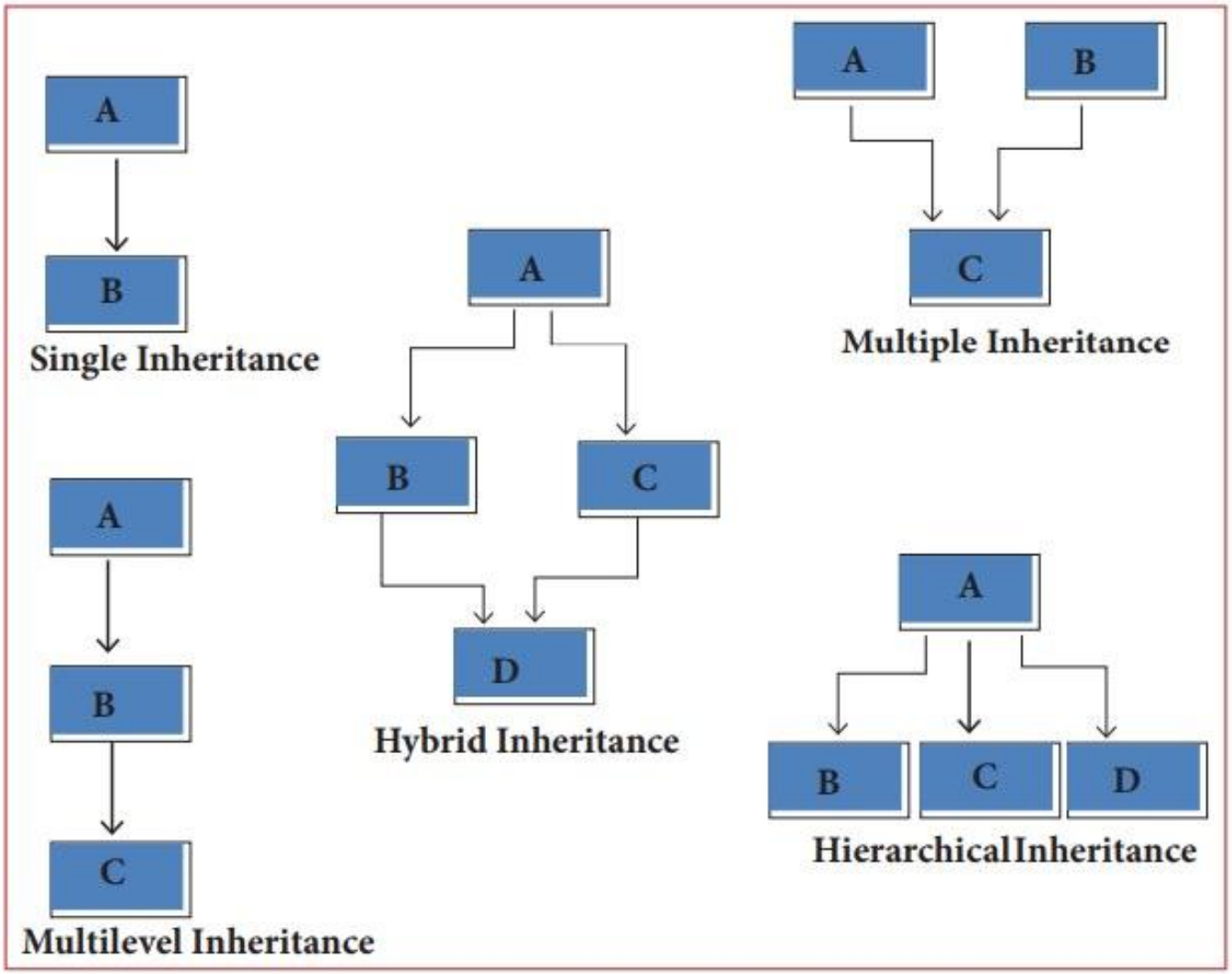
```
Total area: 35
```

- Create a class StudentInfo with name, age and gender and its data members, getInfo() and putInfo() as member functions to get data and display data respectively. Create a derived class studentResult with total (for 5 subjects), percentage and grade as data members and getMarks(), calcGrade() and displayResult() as member functions and inherits from StudentInfo. Get all the details of a student and print the same.

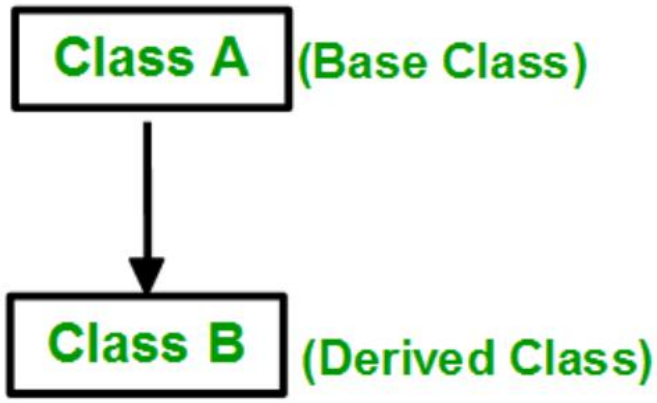
Inheritance – Cont'd

- **Types of 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 Inheritance/ Multipath (also known as Virtual Inheritance)** – a sub class follows multiple types of inheritance while deriving properties from the base or super class

Types of Inheritance



Inheritance – Cont'd
(Single Inheritance in C++)




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

- In this type of inheritance one derived class inherits from only one base class.
- It is the most simplest form of Inheritance.
- All other types of Inheritance are a combination or derivation of Single inheritance.

Single Inheritance - Example

```
#include<iostream>
using namespace std;
class father
{
   public:
    void house()
    {
        cout<<"Have 2BHK House."<<endl;
    }
};
class son:public father
{
    public:
    void car()
    {
        cout<<"Have Audi Car."<<endl;
    }
};
int main()
{
    son o;
    o.house();
    o.car();
    return 0;
}
```

Output:

```
Have 2BHK House.
Have Audi Car.
```

Inheritance – Cont'd (Single Inheritance in C++)

- **Ambiguity in Single Inheritance in C++**
  - If parent and child classes have same named method, parent name and scope resolution operator (::) is used.
  - This is done to distinguish the method of child and parent class since both have same name.

parent and child classes have same named method

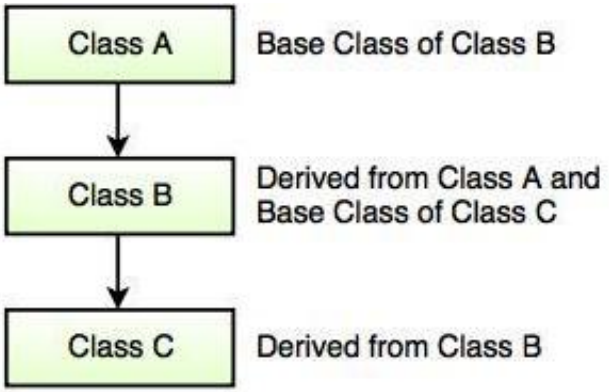
```
#include <iostream>
using namespace std;
class staff
{
    protected:
        string name;
        int code;
    public:
        void getdata();
};
class typist: public staff
{
    private:
        int speed;
    public:
        void getdata();
        void display();
};
void staff::getdata()
{

    cout<<"Name:";
    cin>>name;
    cout<<"Code:";
    cin>>code;
}
```

```
void typist::getdata()
{
    cout<<"Speed:";
    cin>>speed;
}
void typist::display()
{
    cout<<"Name:"<<name<<endl;
    cout<<"Code:"<<code<<endl;
    cout<<"Speed:"<<speed<<endl;
}

int main()
{
    typist t;
    cout<<"Enter data"<<endl;
    t.staff::getdata();
    t.getdata();
    cout<<endl<<"Display data"<<endl;
    t.display();
    return 0;
}
```

Output:

```
Enter data
Name:Ajay
Code:112
Speed:123

Display data
Name:Ajay
Code:112
Speed:123
```

Inheritance – Cont'd
(Multilevel Inheritance in C++)




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

- 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.
- When a class is derived from a class which is also derived from another class, such inheritance is called Multilevel Inheritance.
- **The level of inheritance can be extended to any number of level depending upon the relation.**
- Multilevel inheritance is similar to relation between grandfather, father and child.

Multilevel Inheritance Example

```
#include <iostream>
using namespace std;
class A //Base Class : class A
{
    private:
        int a;
    public:
        void set_a(int val_a)
        {
            a=val_a;
        }
        void disp_a(void)
        {
            cout << "Value of a: " << a << endl;
        }
};
//Here Class B is base class for class C
//and Derived class for class A
class B: public A
{
    private:
        int b;
    public:
        //assign value of a from here
        void set_b(int val_a, int val_b)
        {
            //assign value of a by calling function of class A
            set_a(val_a);
            b=val_b;
        }
        void disp_b(void)
        {
            //display value of a
            disp_a();
            cout << "Value of b: " << b << endl;
        }
};
```

```
//Here class C is derived class and B is Base class
class C: public B
{
    private:
        int c;
    public:
        //assign value of a from here
        void set_c(int val_a, int val_b,int val_c)
        {
            /*** Multilevel Inheritance ***/
            //assign value of a, b by calling function of class B and Class A
            //here Class A is inherited on Class B, and Class B in inherited on Class B
            set_b(val_a,val_b);
            c=val_c;
        }
        void disp_c(void)
        {
            //display value of a and b using disp_b()
            disp_b();
            cout << "Value of c: " << c << endl;
        }
};
int main()
{
    //create object of final class, which is Class C
    C objC;

    objC.set_c(10,20,30);
    objC.disp_c();

    return 0;
}
```

Output:

```
Value of a: 10
Value of b: 20
Value of c: 30
```

Multilevel Inheritance Example

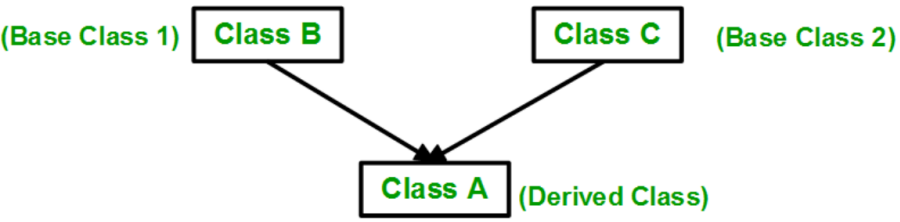
```
#include<iostream>
using namespace std;
class AddData  //Base Class
{
    protected:
        int subjects[3] ;
    public:
        void accept_details()
        {
                cout<<"\n Enter Marks for Three Subjects ";
                cout<<"\n ------------------------------ \n";
                cout<<"\n English : ";
                cin>>subjects[0];
                cout<<"\n Maths : ";
                cin>>subjects[1];
                cout<<"\n History : ";
                cin>>subjects[2];
        }
};
//Class Total - Derived Class.
//Derived from class AddData and Base class of class Percentage
class Total : public AddData
{
    protected:
        int total;
    public:
        void total_of_three_subjects()
        {
                total = subjects[0] + subjects[1] + subjects[2];
        }
};
```

```
//Class Percentage - Derived Class.
// Derived from class Total
class Percentage : public Total
{
    private:
        float per;
    public:
        void calculate_percentage()
        {
                per=total/3.0;
        }
        void show_result()
        {
                cout<<"\n ------------------------------ \n";
                cout<<"\n Percentage of a Student : "<<per;
        }
};
int main()
{
        Percentage p;
        p.accept_details();
        p.total_of_three_subjects();
        p.calculate_percentage();
        p.show_result();
        return 0;
}
```

Output:

```
Enter Marks for Three Subjects
------------------------------

English : 98

Maths : 99

History : 98

------------------------------

Percentage of a Student : 98.3333
```

Inheritance – Cont'd
(Multiple Inheritance in C++)




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

- In C++ programming, a class can be derived from **more than one parents**.
- When a class is derived from **two or more** base classes, such inheritance is called **Multiple Inheritance**.
- Multiple Inheritance in C++ allow us to combine the features of several existing classes into a single class.
- **Syntax:**

class subclass_name : access_mode base_class1, access_mode base_class2, ...
{
  // body of subclass
};

Multiple Inheritance Example

```
#include<iostream>
using namespace std;
class student
{
    protected:
        int rno, m1, m2;
    public:
        void getData()
        {
            cout << "Enter the Roll no :";
            cin>>rno;
            cout << "Enter the two marks    :";
            cin >> m1>>m2;
        }
};
class sports
{
    protected:
        int sm; // sm = Sports mark
    public:
        void getsm()
        {
            cout << "\nEnter the sports mark :";
            cin>>sm;
        }
};
class result : public student, public sports
{
    int tot, avg;
    public:
        void display()
        {
            tot = (m1 + m2 + sm);
            avg = tot / 3;
            cout << "\n\nRoll No: " << rno;
            cout << "\nTotal: " << tot;
            cout << "\nAverage: " << avg;
        }
};
int main()
{
    result obj;
    obj.getData();
    obj.getsm();
    obj.display();
}
```

Output:

```
Enter the Roll no :111
Enter the two marks    :89 98

Enter the sports mark :80


Roll No: 111
Total: 267
Average: 89
```

Ambiguity in Multiple Inheritance

```
class base1
{
  public:
     void someFunction( )
     { .... ... .... }
};
class base2
{
    void someFunction( )
     { .... ... .... }
};
class derived : public base1, public base2
{

};
int main()
{
    derived obj;
    obj.someFunction()// Error!
}
```

This problem can be solved using scope resolution(::) function to specify which function to class either base1 or base2

- In multiple inheritance, a single class is derived from two or more parent classes.
- So, there may be a possibility that two or more parents have **same named member function**.
- If the object of child class needs to access one of the same named member function then it results in **ambiguity**.
- The **compiler is confused** as method of which class to call on executing the call statement.

```
int main()
{
   obj.base1::someFunction( );  // Function of base1 class is called
   obj.base2::someFunction();   // Function of base2 class is called.
}
```

Multiple Inheritance Example

```
#include<iostream>
using namespace std;
class A
{
    protected:
    int x;
    void get()
    {
        cout<<"Enter value of x: ";
        cin >> x;
    }
};
class B
{
    protected:
    int y;
    void get()
    {
        cout<<"Enter value of y: ";
        cin >> y;
    }
};
```

```
//C is derived from class A and class B
class C : public A, public B
{
    public:
    void getData()
    {
        A::get();
        B::get();
    }
    void sum()
    {
        cout << "Sum = " << x + y;
    }
};

int main()
{
     C obj1; //object of derived class C
     obj1.getData();
     obj1.sum();
     return 0;
}
```

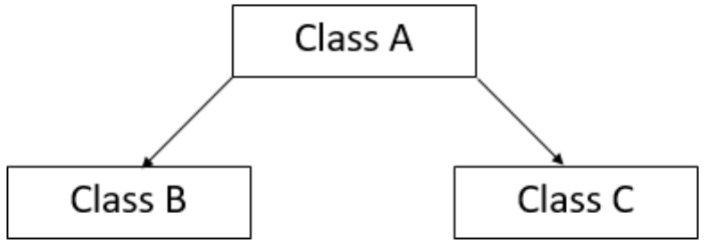
Output:

```
Enter value of x: 4
Enter value of y: 5
Sum = 9
```

- Create two classes named Mammals and MarineAnimals. Create another class named BlueWhale which inherits both the above classes. Now, create a function in each of these classes which prints "I am mammal", "I am a marine animal" and "I belong to both the categories: Mammals as well as Marine Animals" respectively. Now, create an object for each of the above class and try calling
  1 - function of Mammals by the object of Mammal
  2 - function of MarineAnimal by the object of MarineAnimal
  3 - function of BlueWhale by the object of BlueWhale
  4 - function of each of its parent by the object of BlueWhale

Inheritance - Continuation

Inheritance – Cont'd (Hierarchical Inheritance in C++)




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

- Hierarchical Inheritance in C++ is that in which a Base class has many sub classes or when a Base class is used or inherited by many sub classes.
- Thus when more than one classes are derived from a single base class, such inheritance is known as Hierarchical Inheritance,
- *In this inheritance, features that are common in lower levels are included in parent class.*

Hierarchical Inheritance Example

```
#include <iostream>
using namespace std;
class Number
 {
private:
    int num;

public:
    void getNumber(void)
    {
        cout<< "Enter an integer number: ";
        cin>> num;
    }
    int returnNumber(void)
    {
        return num;
    }
};
class Square : public Number
{
public:
    int getSquare(void)
    {
        int num, sqr;
        //get number from class Number
        num = returnNumber();
        sqr = num * num;
        return sqr;
    }
};
```

```
class Cube : public Number
{
public:
    int getCube(void)
    {
        int num, cube;
        //get number from class Number
        num = returnNumber();
        cube = num * num * num;
        return cube;
    }
};
int main()
{
    Square objS;
    Cube objC;
    int sqr, cube;
    objS.getNumber();
    sqr = objS.getSquare();
    cout << "Square of " << objS.returnNumber();
    cout << " is: " << sqr << endl;
    objC.getNumber();
    cube = objC.getCube();
    cout << "Cube   of " << objS.returnNumber();
    cout << " is: " << cube << endl;
    return 0;
}
```

Output:

```
Enter an integer number: 3
Square of 3 is: 9
Enter an integer number: 5
Cube   of 3 is: 125
```

- Create a **base class shape**. Create three different classes, **rectangle**, **circle** and **square** each of which **inherits from the class shape**.
- shape class, has three **attributes**: ***length, breadth, and radius***.
- **The rectangle class** shoul have two methods:
  - *getRectangleDetails()* that prompts the user to enter the length and breadth of the rectangle, and then stores those values in the length and breadth attributes.
  - *rectangle_area()* calculates and returns the area of the rectangle using the formula length * breadth.
- **The circle class** has two methods:
  - *getCircleDetails()* that prompts the user to enter the radius of the circle, and then stores that value in the radius attribute.
  - *circle_area()* calculates and returns the area of the circle using the formula 3.14 * (radius * radius) .
- **The square class** has two methods:
  - *getSquareDetails()* prompts the user to enter the length of one side of the square, and then stores that value in the length attribute.
  - square_area() calculates and returns the area of the square using the formula length * length.
- In main, create objects of the three derived class and test all the functions

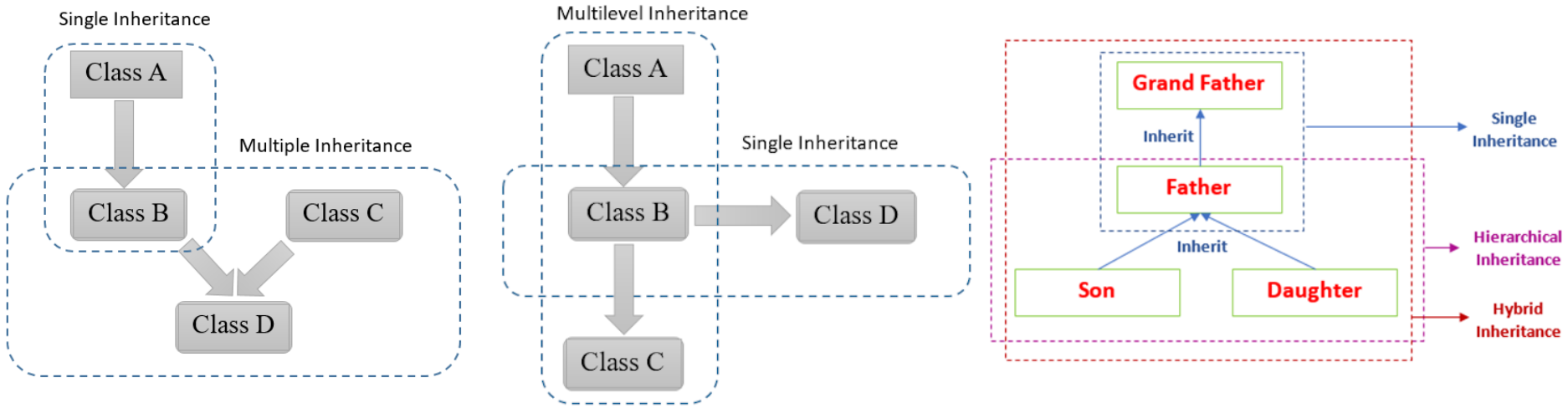
```
#include<iostream>
using namespace std;
class shape
{
protected:
   float length,breadth,radius;
};
class rectangle:public shape
{
   public:
   void getRectangleDetails()
   {
      cout<<"Enter Length: ";
      cin>>length;
      cout<<"Enter Breadth: ";
      cin>>breadth;
   }
   float rectangle_area()
   {
      return length*breadth;
   }
};
class circle:public shape
{
   public:
   void getCircleDetails()
   {
      cout<<"Enter Radius: ";
      cin>>radius;
   }
   double circle_area()
   {
      return 3.14*(radius*radius);
   }
};
 class square:public shape
 {

 public:
    void getSquareDetails()
    {
       cout<<"Enter Side: ";
       cin>>length;
    }
     double square_area()
    {
       return length*length;
    }
};
int main()
{
   rectangle r;
   circle c;
   square s;
   r.getRectangleDetails();
   cout<<"Area of Rectangle   : " ;
   cout<<r.rectangle_area()<<endl;
   c.getCircleDetails ();
   cout<<"Area of Circle   : ";
   cout<<c.circle_area()<<endl;
   s.getSquareDetails ();
   cout<<"Area of Square   : " ;
   cout<<s.square_area()<<endl;
   return 0;
}
```

Output:

```
Enter Length: 4
Enter Breadth: 5
Area of Rectangle   : 20
Enter Radius: 3
Area of Circle   : 28.26
Enter Side: 5
Area of Square   : 25
```

Inheritance – Cont'd
(Hybrid/ Multipath (Virtual) Inheritance in C++)

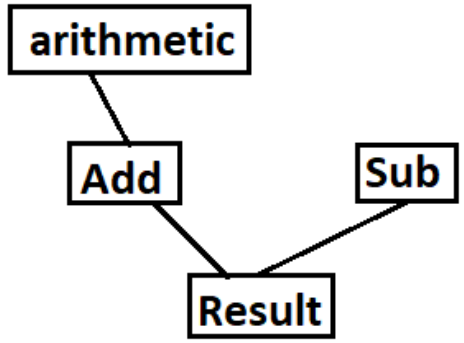
- Hybrid inheritance is a complex form of inheritance in object-oriented programming
- Hybrid inheritance is a combination of more than one type of inheritance (Combining Hierarchical inheritance and Multiple Inheritance.).
- In hybrid inheritance, within the same class, we can have elements of **single inheritance**, **multiple inheritance**, **multilevel inheritance**, and **hierarchical inheritance**.



Hybrid Inheritance Example

```
#include<iostream>
using namespace std;
class arithmetic
{
     protected:
     int num1, num2;
     public:
     void getdata()
     {
          cout<<"For Addition:";
          cout<<"\nEnter the first number: ";
          cin>>num1;
          cout<<"\nEnter the second number: ";
          cin>>num2;
     }
};
class Add:public arithmetic
{
     protected:
     int sum;
     public:
     void add()
     {
          sum=num1+num2;
     }
};
class Sub
{
    protected:
    int n1,n2,diff;
    public:
    void sub()
    {
         cout<<"\nFor Subtraction:";
         cout<<"\nEnter the first number: ";
         cin>>n1;
         cout<<"\nEnter the second number: ";
         cin>>n2;
         diff=n1-n2;
    }
};
```

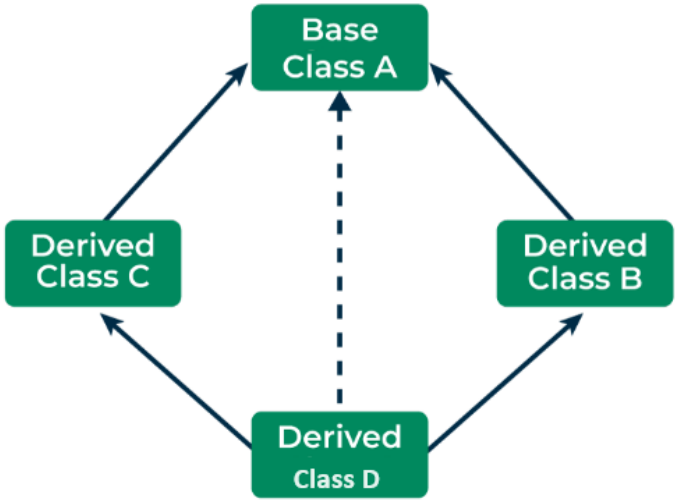
```
class result:public Add, public Sub
{
    public:
    void display()
    {
         cout<<"\nSum of "<<num1<<" and "<<num2<<"= "<<sum;
         cout<<"\nDifference of "<<n1<<" and "<<n2<<"= "<<diff;
    }
};
int  main()
{
    result z;
    z.getdata();
    z.add();
    z.sub();
    z.display();
    return 0;
}
```



Output:

```
For Addition:
Enter the first number: 23

Enter the second number: 45

For Subtraction:
Enter the first number: 89

Enter the second number: 34

Sum of 23 and 45= 68
Difference of 89 and 34= 55
```

Inheritance – Cont'd (Diamond problem in C++)



- The "diamond problem" is a term used in object-oriented programming, particularly in languages like C++ that support multiple inheritance.
- It refers to an issue that arises when a class inherits from two or more classes that have a common base class.
- This common base class can lead to ambiguity in the program because the derived class inherits multiple copies of the common base class, which can result in conflicting or ambiguous method or attribute references.

Diamond problem Example

```
#include<iostream>
using namespace std;
class A
{
public:
    void display()
    {
        cout << "\nA::display()";
    }
};
class B :  public A
{
public:
    void show()
    {
        cout << "\nB::show()";
    }
};
class C :  public A
{
public:
    void output()
    {
        cout << "\nC::output()";
    }
};
class D : public B, public C
{
};
```

```
int main()
{
    D d;
    d.display();
    return 0;
}
```

```
In function 'int main()':
33  error: request for member 'display' is ambiguous
6   note: candidates are: 'void A::display()'
6   note:                 'void A::display()'
=== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```

- In this example, class D inherits from both classes B and C, both of which in turn inherit from class A.
- This is because two instances of class A's display() method is available for class D, one through class B, and the other through class C.
- In this case, the compiler gets confused and cannot decide which name() method it should refer to.

To remove this ambiguity, we use virtual inheritance to inherit the super parent.

Inheritance – Cont'd (Virtual Inheritance)

- Virtual inheritance in C++ is a type of inheritance that **ensures that only one copy or instance of the base class's members is inherited by the grandchild derived class.**
- It is implemented by prefixing the virtual keyword in the inheritance statement.
- **Example:**

```
class B: virtual public A
{
   //members
}
```

- If we now apply virtual inheritance in our previous example, only one instance of class A would be inherited by class D (i.e. B::A and C::A will be treated as the same).

Virtual Inheritance Example

```
#include<iostream>
using namespace std;
class A
{
public:
    void display()
    {
        cout << "\nA::display()";
    }
};
class B : virtual public A
{
public:
    void show()
    {
        cout << "\nB::show()";
    }
};
class C : virtual public A
{
public:
    void output()
    {
        cout << "\nC::output()";
    }
};
class D : public B, public C
{
};
```

```
int main()
{
    D d;
    d.display();
    return 0;
}
```

Output:

```
A::display()
```

Now because there is only one reference of class A's instance is available to the child classes, we get the proper diamond-shaped inheritance.

Create a class named **Shape** with a function that prints "This is a shape". Create another class named **Polygon inheriting the Shape class** with the same function that prints "Polygon is a shape". Create two other classes named **Rectangle and Triangle** having the same function which prints "Rectangle is a polygon" and "Triangle is a polygon" respectively. Again, make another class named **Square** having the same function which prints "Square is a rectangle".

Now, try calling the function by the object of each of these classes.

```
#include <iostream>
using namespace std;
class Shape{
   public:
   Shape(){}
   void print()
   {
      cout<<"\nThis is a shape.";
   }
};
class Polygon: public Shape
{
   public:
   Polygon(){}
   void print()
   {
      cout<<"\nPolygon is a shape.";
   }
};
class Rectangle: public Polygon
{
   public:
   Rectangle(){}
   void print()
   {
      cout<<"\nRectangle is a Polygon.";
   }
};
class Triangle:  public Polygon
{
   public:
   Triangle(){}
   void print()
   {
      cout<<"\nTriangle is a Polygon.";
   }
};
class Square: public Rectangle
{
   public:
   Square(){}
   void print()
   {
      cout<<"\nSquare is a Rectangle.";
   }
};   int main()
     {
        Shape S;
        Polygon P;
        Rectangle R;
        Triangle T;
        Square Sq;
        S.print();
        P.print();
        R.print();
        T.print();
        Sq.print();
        return 0;
     }
```

Output:

```
This is a shape.
Polygon is a shape.
Rectangle is a Polygon.
Triangle is a Polygon.
Square is a Rectangle.
```

UNITIII

OBJECT-ORIENTED PROGRAMMING CONCEPTS

- **Topics to be discussed,**

➢Inheritance

➢**Constructors and Destructors in Derived Classes**

➢Polymorphism and Virtual Functions

Constructors and Destructors in Derived Classes

- Whenever we create an object of a class, the default constructor of that class is invoked automatically to initialize the members of the class.
- If we inherit a class from another class and create an object of the derived class, it is clear that the default constructor of the derived class will be invoked but before that the default constructor of all of the base classes will be invoke, i.e the order of invocation is that the base class's default constructor will be invoked first and then the derived class's default constructor will be invoked.

Constructors in Derived Class Example

```
#include <iostream>
using namespace std;
class Base
{
protected:
   Base()
   {
      cout<<"\nBase class Constructor";
   }
};
class Derived: public Base
{
public:
   Derived()
   {
      cout<<"\nDerived class Constructor";
   }
};

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

Output:

```
Base class Constructor
Derived class Constructor
```

Constructors and Destructors in Derived Classes – Cont'd

- **Why the base class's constructor is called on creating an object of derived class?**
  - when a class is inherited from other, the data members and member functions of base class comes automatically in derived class based on the access specifier but the definition of these members exists in base class only.
  - So when we create an object of derived class, all of the members of derived class must be initialized but the inherited members in derived class can only be initialized by the base class's constructor as the definition of these members exists in base class only.
  - This is why the constructor of **base class is called first to initialize all the inherited members.**

Constructors and Destructors in Derived Classes – Cont'd

- **Order of constructor call for Multiple Inheritance**
  - For multiple inheritance order of constructor call is, the base class's constructors are called in the order of inheritance and then the derived class's constructor.
  - for example if we 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.

Order of constructor call for Multiple Inheritance - Example

```
#include <iostream>
using namespace std;
class A
{
public:
   A()
   {
      cout<<"Class A Constructor\n";
   }
};
class B
{
public:
   B()
   {
      cout<<"Class B Constructor\n";
   }
};
class Derived: public A, public B
{
public:
   Derived()
   {
      cout<<"Derived class Constructor\n";
   }
};
int main()
{
   Derived d;
   return 0;
}
```

Output:

```
Class A Constructor
Class B Constructor
Derived class Constructor
```

Constructors and Destructors in Derived Classes – Cont'd

- **Inheritance in Parameterized Constructor**
  - In the case of the default constructor, it is implicitly accessible from parent to the child class but parameterized constructors are not accessible to the derived class automatically, for this reason, an explicit call has to be made in the child class constructor to access the parameterized constructor of the parent class
- **Syntax Example:**

Derived-Constructor (arg1, arg2, arg3….): Base 1-Constructor (arg1,arg2), Base 2-Constructor(arg3,arg4)

{ …. }

```cpp
#include <iostream>
using namespace std;
class Base
{
protected:
   int x;
public:
   Base(int x)
   {
      this->x=x;
      cout<<"\nBase class Constructor,x="<<x;
   }
};
class Derived: public Base
{
   int y;
public:
   Derived()
   {
        cout<<"\nDerived class Constructor";
         cout<<"\nx="<<x<<"\ny="<<y;
   }
};
int main()
{
   Derived d;
   return 0;
}
```

File	Line	Message
		=== Build file: "no target" in "no project" (compiler: unknown) ===
H:\2024\CS320...		In constructor 'Derived::Derived()':
H:\2024\CS320...	19	error: no matching function for call to 'Base::Base()'
H:\2024\CS320...	8	note: candidate: 'Base::Base(int)'
H:\2024\CS320...	8	note: candidate expects 1 argument, 0 provided
H:\2024\CS320...	3	note: candidate: 'constexpr Base::Base(const Base&)'
H:\2024\CS320...	3	note: candidate expects 1 argument, 0 provided
H:\2024\CS320...	3	note: candidate: 'constexpr Base::Base(Base&&)'
H:\2024\CS320...	3	note: candidate expects 1 argument, 0 provided
		=== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) =

Constructors and Destructors in Derived Classes – Cont'd

- **Important Points**:
  - Whenever the derived class's default constructor is called, the base class's default constructor is called automatically.
  - To call the parameterized constructor of base class inside the parameterized constructor of sub class, we have to mention it explicitly.

Inheritance in Parameterized Constructor - Example

```
#include <iostream>
using namespace std;
class Base
{
   protected:
       int x;
   public:
      Base (int k)   //parameterized constructor of Base class.
     {
        cout<<"\nBase class Parameterized Constructor";
        x = k;
     }
};
class Derived: public Base
{
  int y;
  public:
  Derived(int a, int b):Base(a)  //constructor of child class calling constructor of base class.
  {
     cout<<"\nDerived class Parameterized Constructor";
     y = b;
  }
  void display()
  {
    cout<<"\nx="<<x;
    cout<<"\ny="<<y;
  }
};
```

```
int main()
{
  Derived obj(2,3);
  obj.display();
}
```

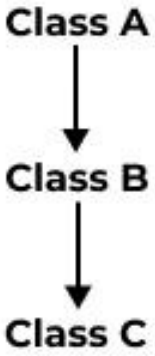
Output:

```
Base class Parameterized Constructor
Derived class Parameterized Constructor
x=2
y=3
```

Constructors and Destructors in Derived Classes – Cont'd

- Destructors in C++ are called in the opposite order of that of Constructors.
- In inheritance, the order of constructors calling is: from *child* class to *parent* class (*child* **->** *parent*).
- In inheritance, the order of constructors execution is: from *parent* class to *child* class (*parent* **->** *class*).
- In inheritance, the order of destructors calling is: from *child* class to *parent* class (*child* **->** *parent*).
- In inheritance, the order of destructors execution is: from *child* class to *parent* class (*child* **->** *parent*).

Order of Calling For Constructors & Destructors in Inheritance



Order of Constructor Call

A() - Class A Constructor

B() - Class B Constructor

C() - Class C Constructor

Order of Destructor Call

C() - Class C Destructor

B() - Class B Destructor

A() - Class A Destructor

Destructors in Derived Classes - Example

```
#include<iostream>
using namespace std;
class baseClass
{
public:
 baseClass()
 {
  cout <<"\nI am baseClass constructor";
 }
  ~baseClass()
 {
  cout <<"\nI am baseClass destructor";
 }
};
class derivedClass: public baseClass
{
public:
 derivedClass()
 {
  cout <<"\nI am derivedClass constructor";
 }
 ~derivedClass()
 {
  cout <<"\nI am derivedClass destructor";
 }
};
```

```
int main()
{
derivedClass D;
return 0;
}
```

Output:

```
I am baseClass constructor
I am derivedClass constructor
I am derivedClass destructor
I am baseClass destructor
```

Constructors and Destructors in Derived Classes – Cont'd

- **Constructor & Destructor in Multiple inheritance**

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

  - Here, A class in inherited first, so constructor of class A is called first then the constructor of class B will be called next.
  - The destructor of derived class will be called first then destructor of base class which is mentioned in the derived class declaration is called from last towards first sequence wise.

Constructor & Destructor in Multiple inheritance

```
#include<iostream>
using namespace std;
class baseClass1
{
 public:
  baseClass1()
  {
   cout<<"\nI am baseClass1 constructor";
  }
  ~baseClass1()
  {
   cout<<"\nI am baseClass1 destructor";
  }
};
class baseClass2
{
public:
 baseClass2()
 {
  cout<<"\nI am baseClass2 constructor";
 }
 ~baseClass2()
 {
  cout<<"\nI am baseClass2 destructor";
 }
};
class derivedClass: public baseClass1, public baseClass2
{
  public:
   derivedClass()
   {
    cout<<"\nI am derivedClass constructor";
   }
   ~derivedClass()
   {
    cout<<"\nI am derivedClass destructor";
   }
};
 int main()
 {
 derivedClass D;
 return 0;
}
```

Output:

```
I am baseClass1 constructor
I am baseClass2 constructor
I am derivedClass constructor
I am derivedClass destructor
I am baseClass2 destructor
I am baseClass1 destructor
```

Object Composition in C++

- Composition is referred to building a complex thing with the use of smaller and simple parts.
- For example,
  - A car is built using a metal frame, an engine some tires, a transmission system, a steering wheel, and large number of other parts.
  - A personal computer is built from a CPU, a motherboard, memory unit, input and output units etc.
- Composition is one of the fundamental approaches or concepts used in object-oriented programming.
- This process of building complex objects from simpler ones is called **object composition**.

Object Composition in C++ - Cont'd

- Broadly speaking, object **composition models** a **"has-a" relationship** between two objects.
- A car "has-a" tyre, computer "has-a" CPU etc.
- The complex object is sometimes called the whole, or the parent.
- The simpler object is often called the part, child, or component.
- Object Composition is useful in a C++ context because it allows us to create complex classes by combining simpler, more easily manageable parts.
- This reduces complexity, and allows us to write code faster and with less errors because we can reuse code that has already been written, tested, and verified as working.

Object Composition in C++- Cont'd

Syntax:

```
class A
{
   // body of a class
};
class B
{
   A objA;
   public:
  B(arg-list) : objA(arg-list1);
};
```

simple class

complex class

- In the classes given above, B uses objects of class A as its data members.
- Hence, B is a complex class that uses a simple class A.

Object Composition - Example

```
#include <iostream>
using namespace std;
 // Simple class
class A
{
public:
   int x;
   A() { x = 0; }
   A(int a)
   {
      cout << "Constructor A(int a) is invoked\n";
      x = a;
   }
};
```

```
// Complex class
class B
{
   int data;
   A objA;
 public:
   B(int a) : objA(a)
   {
      data = a;
   }
    void display()
   {
      cout << "Data in object of class B = " << data
         << endl;
      cout << "Data in member object of "
         << "class A in class B = " << objA.x;
   }
};
int main()
{
   B objb(25);
   objb.display();
   return 0;
}
```

Output:

```
Constructor A(int a) is invoked
Data in object of class B = 25
```

Object Composition in C++ - Cont’d

- **Types of Object Composition in C++**
  - Object composition is basically of the following subtypes:
    - **Composition**
    - **Aggregation**
    - **Object Delegation**

Object Composition in C++- Cont'd

- **Composition:**
  - **Composition** relationship is also called a **part-whole** relationship in which the part component can only be a part of a single object simultaneously.
  - **In composition relationships, the part component will be created when the object is created, and the part will be destroyed when the object is destroyed.**
  - A person's body and heart is a good example of a part-whole relationship where if a heart is part of a person's body, then it cannot be a part of someone else's body at one time.

Object Composition in C++- Cont'd

- To qualify as a composition, the object and a part must have the following relationship-
  - The part (member) is part of the object (class).
  - The part (member) can only belong to one object (class).
  - The part (member) has its existence managed by the object (class).
  - The part (member) does not know about the existence of the object (class).
- There are some variations on the rule of creating and destroying parts:
  - A composition may avoid creating some parts until they are needed.
  - A composition may opt to use a part that has been given to it as input rather than creates the part itself.
  - A composition may delegate the destruction of its parts to some other object.

Object Composition in C++- Cont'd

- **Aggregation:**
  - The aggregation is also a part-whole relationship but here in aggregation, the parts can belong to more than one object at a time, and the whole object is not responsible for the existence of the parts.
  - To qualify as aggregation, a whole object and its part must have the following relationships:
    - The part (member) is part of the object (class).
    - The part (member) can belong to more than one object (class) at a time.
    - The part (member) does not have its existence managed by the object (class).
    - The part (member) does not know about the existence of the object (class).

Object Composition in C++- Cont'd

- **Object Delegation :**
  - **Object delegation** is a process in which we use the objects of a class as a member of another class.
  - Object delegation is the passing of work from one object to another.
  - It is an alternative to the process of inheritance.
  - But when the concept of inheritance is used in the program, it shows an **is-a** relationship between two different classes.
  - On the contrary, in object delegation, there is no relationship between different classes.

Object Delegation - Example

```
#include <iostream>
using namespace std;
class First
{
public:
   void print()
   {
      cout << "class First print method";
   }
};
class Second
{
   First fobj;
 public:
   void print()
    {
      fobj.print();
   }
};
```

```
int main()
{
   Second sobj;
   sobj.print();
   return 0;
}
```

Output:

```
class First print method
```

UNITIII

OBJECT-ORIENTED PROGRAMMING CONCEPTS

- **Topics to be discussed,**

➢ Inheritance

➢ Constructors and Destructors in Derived Classes

➢ **Polymorphism and Virtual Functions**

Polymorphism and Virtual Functions

- **Polymorphism** is one of the **most important concepts of Object-Oriented Programming (OOPs)**.
- We can describe the word *polymorphism as an object having many forms*.
- Polymorphism is the notion that can hold up the ability of an object of a class to show different responses.
- In other words, we can say that polymorphism is the ability of an object to be represented in over one form.
- To understand polymorphism, we can consider a real-life example. We can relate it to the relationship of a person with different people. A man can be a father to someone, a husband, a boss, an employee, a son, a brother, or can have many other relationships with various people. Here, this man represents the object, and his relationships display the ability of this object to be represented in many forms with totally different characteristics.
- Polymorphism in C++ can be broadly categorized into two types :
  - Compile-time Polymorphism
  - Runtime Polymorphism

Polymorphism and Virtual Functions

- **Compile-time Polymorphism:**
  - It is called *early binding* or *static binding*.
  - We can implement compile-time polymorphism using function overloading and operator overloading.
  - Method/function overloading is an implementation of compile-time polymorphism where the same name can be assigned to more than one method or function, having different arguments or signatures and different return types. (discussed earlier).
- **Runtime Polymorphism:**
  - In runtime polymorphism, the compiler resolves the object at run time and then it decides which function call should be associated with that object.
  - It is also known as *dynamic or late binding polymorphism*.
  - This type of polymorphism is executed through virtual functions and function overriding.
  - All the methods of runtime polymorphism get invoked during the run time.

Polymorphism and Virtual Functions –Cont'd (Function Overriding in C++)

- **When a derived class or child class defines a function that is already defined in the base class or parent class, it is called function overriding in C++.**
- The new function definition in the derived class must have the **same function name** and **same parameter list** as in the base class.
- Function overriding helps us achieve runtime polymorphism and the enables programmers to perform the specific implementation of a function already used in the base class.
- In this scenario, the member function in the base class is called the **overridden function** and the member function in the derived class is called the **overriding function**. There must be an **IS-A relationship** (i.e. **inheritance**).

Function Overriding -Example

```
#include <iostream>
using namespace std;
class Base// Base class
{
   public:
      void print()// base member function (overridden function)
      {
         cout << "\nprint function of base class";
      }
};
// Derived class
class Derived : public Base
{
    public:
      void print()// derived member function (overriding function)
      {
         cout << "\nprint function of derived class";
      }
};
int main()
{
   Derived dobj;
   dobj.print(); // calling overriding function
   return 0;
}
```

Output:

```
print function of derived class
```

Polymorphism and Virtual Functions –Cont'd (Function Overriding in C++)

Working of the Function Overriding Principle

```
class Base {
    public:
      void print() {
          // code
      }
};

class Derived : public Base {
    public:
      void print() {
          // code
      }
};

int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```

- In the previous example, the function print() is declared in both the Base and Derived classes.
- When we call the function print() through the Derived class object, "dobj", the print() from the Derived class is invoked and executed by overriding the same function of the Base class.

As we can see from this image, the Base class function was overridden because we called the same function through the object of the Derived class.

Polymorphism and Virtual Functions –Cont'd (Function Overriding in C++)

- If we call the print() function through an object of the Base class, the function will not be overridden.
- For Example,

```
//Call function of Base class
Base base1;
base1.print();
```

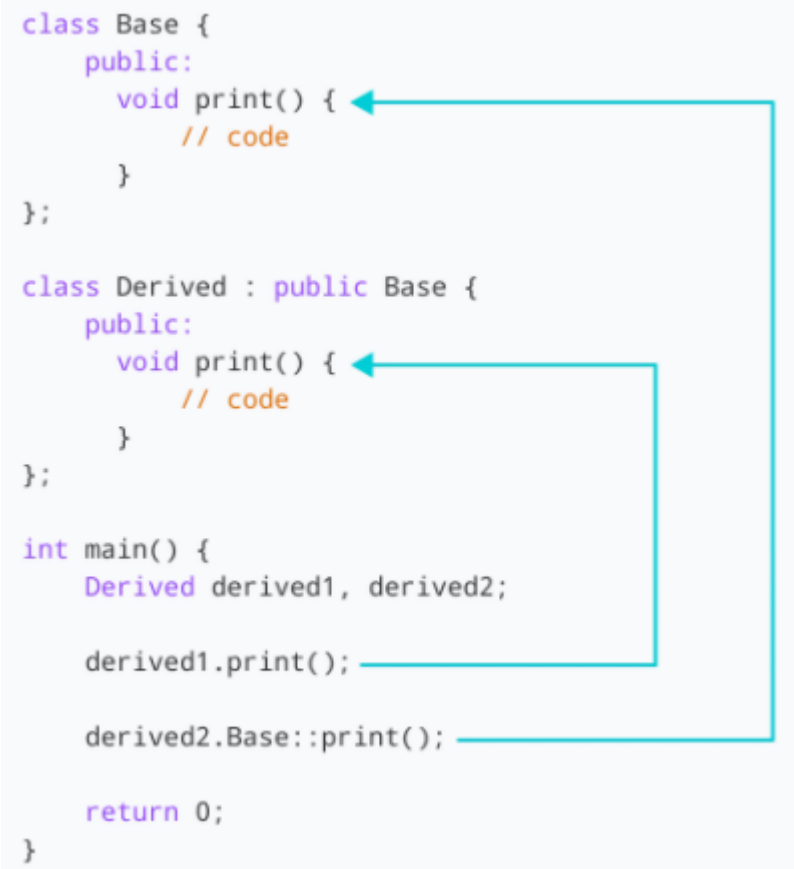
- The output of the above code will be:
  - print function of base class
- **To access Overridden Functions in C++**
  - we must use the scope resolution operator, "::" to access the overridden function.
  - Another way to access the overridden function is by using the pointer of the base class to point to an object of the derived class and calling the function through the pointer.

```
using namespace std;
class Base// Base class
{
   public:
      void print()// base member function (overridden function)
      {
         cout << "\nprint function of base class";
      }
};
// Derived class
class Derived : public Base
{
    public:
      void print()// derived member function (overriding function)
      {
         cout << "\nprint function of derived class";
      }
};
int main()
{
   Base bobj;
   bobj.print();//calling base class method
   Derived dobj;
   dobj.print(); // calling overriding function
   dobj.Base::print();//calling base class method
}
```

Output:

```
print function of base class
print function of derived class
print function of base class
```

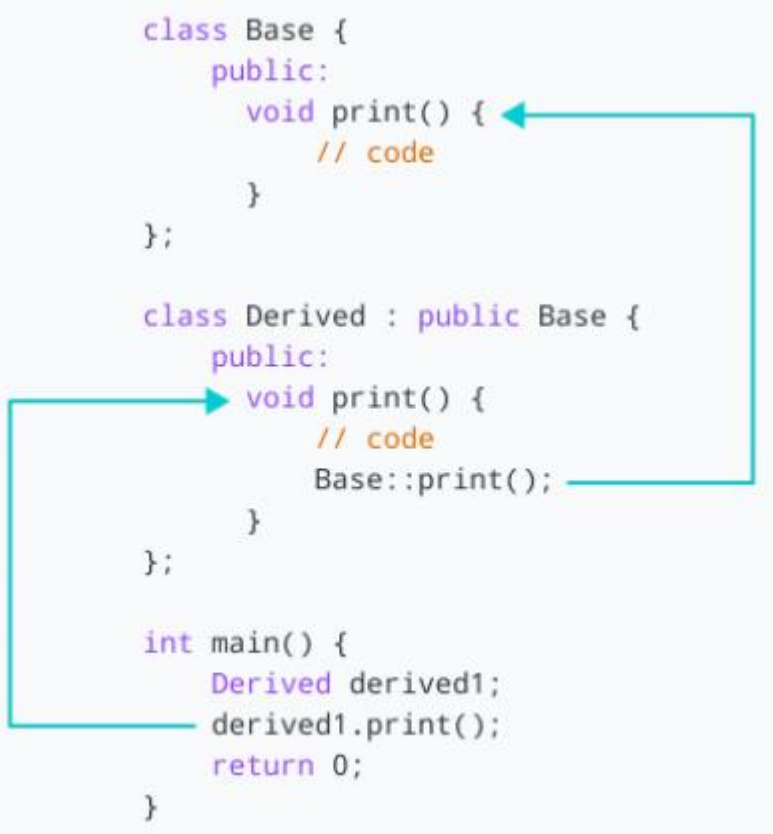
Polymorphism and Virtual Functions – Cont'd



```
class Base {
    public:
      void print() {
          // code
      }
};

class Derived : public Base {
    public:
      void print() {
          // code
      }
};

int main() {
    Derived derived1, derived2;

    derived1.print();

    derived2.Base::print();

    return 0;
}
```

- ***Working of the Access of overridden function***
  - Here the statement derived 1.print() accesses the print() function of the Derived class and the statement derived2.Base::print() accesses the print() function of the Base class.

Polymorphism and Virtual Functions –Cont'd



- **Calling a C++ overridden function from the derived class**
  - In this code, we call the overridden function from within the Derived class itself.

Polymorphism and Virtual Functions –Cont'd

Difference Between Function Overloading and Overriding in C++

Function Overloading	Function Overriding
Function Overloading **provides multiple definitions of the function** by changing signature.	Function Overriding is the **redefinition of base class function** in its derived class with same signature.
An example of compile time polymorphism.	An example of run time polymorphism.
Function signatures should be different.	Function signatures should be the same.
Overloaded functions are in same scope.	Overridden functions are in different scopes.
Overloading is used when the same function has to behave differently depending upon parameters passed to them.	Overriding is needed when derived class function has to do some different job than the base class function.
A function has the ability to load multiple times.	A function can be overridden only a single time.
In function overloading, we don't need inheritance.	In function overriding, we need an inheritance concept.

Dynamic Binding

- In case of few programs**, it is impossible to know which function is to be called until run time. This is called dynamic binding**
- Dynamic binding can be implemented with function pointers.
- In this method, the pointer points to a function instead of a variable.

Dynamic Binding - Example

```
#include <iostream>
using namespace std;
float add(float x,float y)
{
   return x+y;
}
float sub(float x,float y)
{
   return x-y;
}
float mul(float x,float y)
{
   return x*y;
}
float div(float x,float y)
{
   return x/y;
}
float (*ptr)(float,float);
```

```
int main()
{
   int ch;
   float n1,n2,res;
   cout<<"\nEnter 2 numbers:";
   cin>>n1>>n2;
   do
   {
      cout<<"1.Add\n";
      cout<<"2.subtract\n";
      cout<<"3.Multiply\n";
      cout<<"4.divide\n";
      cout<<"Enter your choice:";
      cin>>ch;
   }while(ch<1||ch>4);
   switch(ch)
      {
         case 1:ptr=add;break;
         case 2:ptr=sub;break;
         case 3:ptr=mul;break;
         case 4:ptr=div;break;
      }
      cout<<"Result="<<ptr(n1,n2);
}
```

In this program, instead of calling functions directly, we have called them through function pointer. The compiler is unable to use static or early binding in this case. In this program the compiler has to read the addresses held in the pointers toward different functions. Until runtime, decisions are not taken as to which function needs to be executed, hence it is late binding.

Output:

```
Enter 2 numbers:5
12
1.Add
2.subtract
3.Multiply
4.divide
Enter your choice:1
Result=17
```

Pointers to derived objects

- **Pointers to objects of a base class are type-compatible with pointers to objects of a derived class.**
- Therefore, a single pointer variable can be made to point to objects belonging to different classes.
- If **B is base class** and **D is derived class** from B, then

```
B *bptr; // Pointer to base class
B bobj; //Base class object
D dobj; //Derived class object
bptr=&bobj // base pointer points to base object

bptr=&dobj // base pointer points to derived object
```

  - This is perfectly valid with C++ because d is an object derived from B.

Pointers to derived objects

- **Base class pointer can access only those members which are inherited from B and not the members that originally belong to D**
- **In case a member of D has the same name as one of the members of B, then any reference to that member by bptr will always access the base class member.**

```
#include <iostream>
using namespace std;
class B
{
public:
   int b;
   void display()
   {
      cout<<"\nBase class display, b="<<b;
   }
};
class D:public B
{
public:
   int d;
   void display()
   {
      cout<<"\nDerived class display, b="<<b<<" d="<<d;
   }
};
```

```
int main()
{
   B *bptr,bobj;
   bptr=&bobj;
   bptr->b=100;
   bptr->display();
}
```

```
Base class display, b=100
```

```
#include <iostream>
using namespace std;
class B
{
public:
   int b;
   void display()
   {
      cout<<"\nBase class display, b="<<b;
   }
};
class D:public B
{
public:
   int d;
   void display()
   {
      cout<<"\nDerived class display, b="<<b<<" d="<<d;
   }
};
```

```
21  int main()
22  {
23     B *bptr;
24     D dobj;
25     bptr=&dobj;
26     bptr->b=200;
27     bptr->d=300;
28     bptr->display();
29  }
```

File	Line	Message
		=== Build file: "no target" in "no project" (compiler: unknown) ===
H:\2024\CS320...		In function 'int main()':
H:\2024\CS320...	27	error: 'class B' has no member named 'd'
		=== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===

Base class pointer can access only those members which are inherited from B and not the members that originally belong to D

Polymorphism and Virtual Functions –Cont'd

Overriding a non-virtual function

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

```
using namespace std;
class Base
{
    public:
    void print()
    {
        cout<<"\nBase class print method";
    }
};
class Derived: public Base
{
    public:
    void print()
    {
        cout<<"\nDerived class print method";
    }
};
int main()
{
  Base *bobj;
  Derived dobj;
  bobj->print();
  bobj=&dobj;
  bobj->print();
}
```

In case a member of D has the same name as one of the members of B, then any reference to that member by bptr will always access the base class member.

Output:

```
Base class print method
Base class print method
```

```
#include<iostream>
using namespace std;
class Base
{
   public:
      int b;
   void print()
   {
      cout<<"\nBase class print method, b="<<b;
   }
};
class Derived: public Base
{
   public:
   int d;
   void print()
   {
      cout<<"\nDerived class print method,b="<<b<<" d="<<d;
   }
};
```

```
int main()
{
  Base *bptr,bobj;
  bptr=&bobj;
  bptr->b=1;
  bptr->print();
  Derived dobj;
  bptr=&dobj;
  ((Derived *)bptr)->b=100;
  ((Derived *)bptr)->d=200;
  ((Derived *)bptr)->print();
}
```

```
Base class print method, b=1
Derived class print method,b=100 d=200
```

Polymorphism and Virtual Functions –Cont'd

Virtual Functions

- **Dynamic binding of member functions in C++ can be done using virtual keyword**
- A **virtual function** is a C++ member function which is declared within a base class and is overridden (redefined) by a derived class.
- It is achieved by using the keyword 'virtual' in the base class.
- When we refer to a derived class object using a pointer (or reference) to the base class, we can call a virtual function for that object and execute the derived class's version of the function.

Polymorphism and Virtual Functions –Cont'd

Virtual Functions

- **Some properties of the virtual functions are mentioned below**
  - Virtual functions assure that the correct function is to be invoked (i.e. called) for an object, irrespective of the type of pointer (or reference) used for the function call.
  - They are primarily used to achieve runtime polymorphism.
  - Functions are declared with the virtual keyword in the base class.
  - The function call is resolved at runtime.

Polymorphism and Virtual Functions –Cont'd
Virtual Functions

- **Rules for Virtual Functions:**
  - Virtual functions cannot be static and friend to another class
  - Virtual functions must be accessed using pointers or references of base class type
  - The function prototype should be same in both base and derived classes
  - A class must not have a virtual constructor. But it can have a virtual destructor
  - They are always defined in the base class and redefined in the derived class

Virtual Function Example

```
#include<iostream>
using namespace std;
class Base
{
    public:
    virtual void print()
    {
        cout<<"\nBase class print method";
    }
    void show()
    {
        cout<<"\nBase class show method";
    }
};
class Derived: public Base
{
    public:
    void print()
    {
        cout<<"\nDerived class print method";
    }
    void show()
    {
        cout<<"\nDerived class show method";
    }
};
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:

```
Derived class print method
Base class show method
```

Polymorphism and Virtual Functions –Cont'd

Virtual Functions

- **Runtime polymorphism is achieved only through a pointer (or reference) of base class type.**
- Also, a base class pointer can point to the objects of the base class as well as to the objects of the derived class.
- In the previous code, base class pointer 'bptr' contains the address of object 'd' of the derived class.
- **Late binding(Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer) and Early binding (Compile-time) is done according to the type of pointer**
- Since print() function is declared with the virtual keyword so it will be bound at run-time (output is "Derived class print method" as a pointer is pointing to object of derived class)
- show() is non-virtual so it will be bound during compile time(output is "Base class show method "as a pointer is of base type).

Polymorphism and Virtual Functions –Cont'd

Pure Virtual Functions and Abstract Classes in C++

- Sometimes implementation of all functions cannot be provided in a base class because we don't know the implementation.
- Such a class is called an **abstract class**.
  - For example, let Shape be a base class.
  - We cannot provide the implementation of function draw() in Shape, but we know every derived class must have an implementation of draw().
  - We cannot create objects of abstract classes.
- A **pure virtual function** (or abstract function) in C++ is a virtual function that is declared in the base class but we cannot implement it, with a '0' assigned to make them pure.
- In this way, the base class becomes an abstract class, and it must be inherited by a derived class, which provides an implementation for it.
- **Syntax:** virtual Return_type function_name() = 0;

Polymorphism and Virtual Functions –Cont'd

Pure Virtual Functions and Abstract Classes in C++

- **Characteristics of Pure virtual functions:**
  - These functions also must have a prefix before the function's name called 'virtual'.
  - In the base class, we can declare it, but we cannot implement it.
  - '0' must be assigned to the function to make them pure.
  - The derived class must provide the implementation code for this function, else this derived class is also termed an 'abstract class'.
  - Note that classes having at least one pure virtual function are called Abstract classes.
- **The main use of pure virtual functions is to create an abstract class that defines an interface for its derived classes.**

Polymorphism and Virtual Functions –Cont'd

Pure Virtual Functions and Abstract Classes in C++

- *An abstract class is a class in C++ which have at least one pure virtual function*.
- **An abstract class can have normal functions and variables along with a pure virtual function.**
- **An abstract class cannot be instantiated, but pointers and references of Abstract class type can be created**
- Abstract classes are mainly used for Upcasting so that its derived classes can use their interface
- **If an Abstract Class has derived class, they must implement all pure virtual functions, or else they will become Abstract too**
- **An abstract class is like a base class for other classes to provide a common interface to implement. This helps us use the polymorphism feature of the programming language**.

Pure Virtual Function Example

```
#include<iostream>
#include<cmath>
using namespace std;
class Shape  // Abstract class
{
    public:
    // calcArea is a pure virtual function
    virtual float calcArea() = 0;
};
class Square : public Shape
{
    int a;
    public:
    Square(int l)
    {
        a = l;
    }
    // Calculates and returns area of square
    float calcArea()
    {
        return static_cast<float>(a*a);
    }
};
class Circle : public Shape
{
    int r;
    public:
    Circle(int x)
    {
        r = x;
    }
    // Calculates and returns area of circle
    float calcArea()
    {
        return static_cast<float>(M_PI*r*r) ;
    }
};
class Rectangle : public Shape
{
    int l;
    int b;
    public:
    Rectangle(int x, int y)
    {
       l=x;
       b=y;
    }
    // calculates and returns the area of rectangle
    float calcArea()
    {
        return static_cast<float>(l*b);
    }
};
int main()
{
    Shape *shape;
    Square s(4);
    Rectangle r(5,6);
    Circle c(7);
    shape =&s;
    float a1 =shape->calcArea();
    shape = &r;
    float a2 = shape->calcArea();
    shape = &c;
    float a3 = shape->calcArea();
    std::cout << "\nThe area of square is: " <<a1;
    std::cout << "\nThe area of  rectangle is: " <<a2;
    std::cout << "\nThe area of  circle is: " <<a3;
    return 0;
}
```

Output:

```
The area of square is: 16
The area of  rectangle is: 30
The area of  circle is: 153.938
```

Polymorphism and Virtual Functions –Cont’d

Virtual Function	Pure Virtual Function
In the virtual function, the derived class overrides the function of the base class; it is the case of the function overriding.	In a pure virtual function, the derived call would not call the base class function as it has not defined instead it calls the derived function which implements that same pure virtual function in the derived call.
Class containing virtual function may or may not be an Abstract class.	If there is any pure virtual function in a class, then it becomes an "Abstract class".
Virtual function in the base does not enforce to derived for defining or redefining	In pure virtual function, the derived class must redefine the pure virtual class of the base class. Otherwise, that derived class will become abstract as well.

Polymorphism and Virtual Functions –Cont'd

- **Advantages of Pure Virtual Functions**
  - Abstraction: Pure virtual functions are a way to separate the interface from the implementation, to make the code easier to maintain.
  - Polymorphism: A base class pointer is used to call functions of its derived classes, a key way to use polymorphism in C++.
  - Reusability: Since we define a common interface, we reduce the amount of code and make it more reusable.

Polymorphism and Virtual Functions –Cont'd

Need for Virtual Destructors

- Destructors of the class can be declared as virtual.
- Whenever we do upcast i.e. assigning the derived class object to a base class pointer, the ordinary destructors can produce unacceptable results.

```
#include<iostream>
using namespace std;
class Base
{
 public:
   Base()
   {
     cout<<"\nBase Class::Constructor";
   }
   ~Base()
   {
     cout<<"\nBase Class::Destructor";
   }
};
class Derived: public Base
{
 public:
  Derived()
   {
     cout<<"\nDerived class::Constructor";
   }
   ~Derived()
   {
     cout<<"\nDerived class::Destructor";
   }
};
int main()
{
 Base * b = new Derived;// Upcasting
 delete b;
}
```

Output:

```
Base Class::Constructor
Derived class::Constructor
Base Class::Destructor
```

- In this program, Ideally, the destructor that is called when "delete b" is called should have been that of derived class but we can see from the output that destructor of the base class is called as base class pointer points to that.
- Due to this, the derived class destructor is not called and the derived class object remains intact thereby resulting in a memory leak.
- The solution to this is to make base class constructor virtual so that the object pointer points to correct destructor and proper destruction of objects is carried out.

Virtual Destructor Example

```
#include<iostream>
using namespace std;
class Base
{
 public:
   Base()
   {
     cout<<"\nBase Class::Constructor";
   }
   virtual ~Base()
   {
     cout<<"\nBase Class::Destructor";
   }
};
class Derived: public Base
{
 public:
  Derived()
   {
     cout<<"\nDerived class::Constructor";
   }
   ~Derived()
   {
     cout<<"\nDerived class::Destructor";
   }
};
int main()
{
 Base * b = new Derived;// Upcasting
 delete b;
}
```

Output:

```
Base Class::Constructor
Derived class::Constructor
Derived class::Destructor
Base Class::Destructor
```

Difference Between Compile Time And Run Time Polymorphism

Compile-Time Polymorphism	Run-Time Polymorphism
It is also called Static Polymorphism.	It is also known as Dynamic Polymorphism.
In compile-time polymorphism, the compiler determines which function or operation to call based on the number, types, and order of arguments.	In run-time polymorphism, the decision of which function to call is determined at runtime based on the actual object type rather than the reference or pointer type.
Function calls are statically binded.	Function calls are dynamically binded.
Compile-time Polymorphism can be exhibited by: 1. Function Overloading 2. Operator Overloading	Run-time Polymorphism can be exhibited by Function Overriding.
Faster execution rate.	Comparatively slower execution rate.
Inheritance in not involved.	Involves inheritance.