

# UNIT III

## 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 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
    }
};

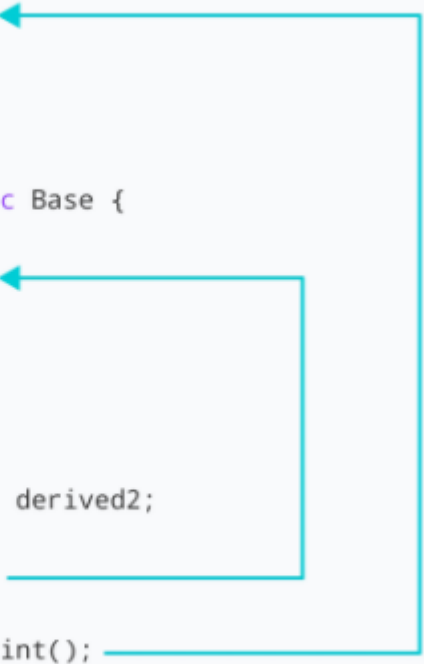
class Derived : public Base {
public:
    void print() {
        // code
    }
};

int main() {
    Derived derived1, derived2;

    derived1.print();

    derived2.Base::print();

    return 0;
}
```



- ***Working of the Access of overridden function***

- Here the statement derived1.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

```
class Base {  
    public:  
    void print() {  
        // code  
    }  
};  
  
class Derived : public Base {  
    public:  
    void print() {  
        // code  
        Base::print();  
    }  
};  
  
int main() {  
    Derived derived1;  
    derived1.print();  
    return 0;  
}
```

- **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 <b>provides multiple definitions of the function</b> by changing signature.	Function Overriding is the <b>redefinition of base class function</b> in its derived class with same signature.
An example of <b>compile time polymorphism</b> .	An example of <b>run time polymorphism</b> .
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
H:\2024\CS320...		=== Build file: "no target" in "no project" (compiler: unknown) ===
H:\2024\CS320...	27	In function 'int main()': 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 <b>Static Polymorphism</b> .	It is also known as <b>Dynamic Polymorphism</b> .
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 is not involved.	Involves inheritance.