# Lab 9 – Polymorphism

---

**Lab Outcomes:**

- **Understanding Polymorphism through Virtual Functions**: Comprehend the concept of polymorphism and its implementation in C++ using virtual functions. Gain the ability to use virtual functions to allow derived classes to provide specific implementations that differ from the base class, enabling flexible and extensible code.
- **Exploring Early and Late Binding**: Distinguish between early (compile-time) and late (run-time) binding in C++. Understand how the virtual keyword facilitates late binding, allowing the program to determine the appropriate function to call based on the object's actual type during runtime.
- **Practical Use of Function Overloading**: Learn to implement function overloading within a single class by defining multiple functions with the same name but differing in parameter types or numbers. This outcome helps demonstrate how C++ handles multiple definitions within a class scope.
- **Applying Function Overriding in Derived Classes**: Master function overriding by defining functions with the same signature in base and derived classes. Recognize how function overriding facilitates polymorphism and supports code reusability by allowing derived classes to modify inherited behaviors.
- **Working with Base Class Pointers for Derived Class Objects**: Utilize base class pointers to call virtual functions of derived classes. This outcome reinforces the application of polymorphism, as students observe how base class pointers can invoke functions from different derived classes based on the runtime type of the object being pointed to.

---

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

For example, think of a base class called Animal that has a method called animalSound(). Derived classes of Animals could be Cats, Dogs, Birds - And they also have their own implementation of an animal sound, and the cat meows, etc).

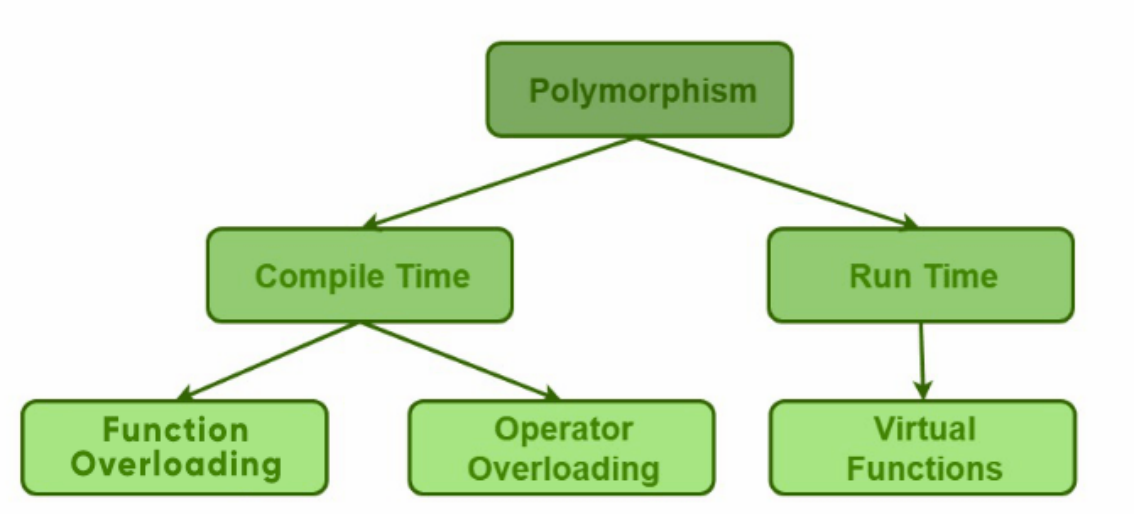
To realize polymorphism in C++, you need

- A virtual function and
- A pointer of base class type

A member function of a base class can be made a virtual function if it is probable to redefine (override) this function in the derived class. The significance of making a function virtual becomes evident when you want to invoke function of derived class from a pointer of base class referencing derived class's object.

## Types of Polymorphism

- **Compile-time Polymorphism**
- **Runtime Polymorphism**




# 1.Compile Time Polymorphism

This type of polymorphism is achieved by function overloading or operator overloading.

# Function Overloading

Function overloading is the availability of various functions within a class that differ from each other in function signature i.e. various functions share same name with different parameter types or number of parameters. Inheritance is not necessarily required to overload functions within a program. On the other hand, in function overriding, there must exists inheritance relationship between classes (i.e. base and derived) and functions' signature are also required to be same in both parent and child class(es). However, derived class(es) redefine function(s) having signature similar to base class's function(s).
Following code provides an example where both concepts (function overloading and overriding) have been elaborated:

```
class Base {

  protected:

        void myFunc() {
        cout<<"Base Class' Function";
        }
};
class Derived: public Base {
 public:
        void myFunc()
        {
        cout<<"Derived Class' Function";
        }
        void myFunc(int a)
```

```
{
cout<<"Derived Class' Function with Parameter Value" <<a; } };
```

- At first, the class *Derived* shows function overloading while containing two functions differ from each other in function signature (i.e. functions *myFunc()* and *myFunc(int)*).
- Secondly, it also shows function overriding as it holds two function implementations with same signature (i.e. besides *myFunc()* of its own, it also contains *myFunc()* inherited from the Base class.

```cpp
// C++ program to demonstrate
// function overloading or
// Compile-time Polymorphism
#include <bits/stdc++.h>

using namespace std;
class Geeks {
public:
    // Function with 1 int parameter
    void func(int x)
    {
        cout << "value of x is " << x << endl;
    }

    // Function with same name but
    // 1 double parameter
    void func(double x)
    {
        cout << "value of x is " << x << endl;
    }

    // Function with same name and
    // 2 int parameters
    void func(int x, int y)
    {
        cout << "value of x and y is " << x << ", " << y
             << endl;
    }
};

// Driver code
int main()
{
    Geeks obj1;

    // Function being called depends
    // on the parameters passed
    // func() is called with int value
    obj1.func(7);

    // func() is called with double value
    obj1.func(9.132);

    // func() is called with 2 int values
    obj1.func(85, 64);
    return 0;
}
```

Output

```
value of x is 7
value of x is 9.132
value of x and y is 85, 64
```

**Explanation:** In the above example, a single function named function **func()** acts differently in three different situations, which is a property of polymorphism.

```
#include<iostream>
using namespace std;
class A{
    public:
        void f1(){
        }
        void f2(){
        cout<<"Inside A";
        }
};
class B:public A{
    public:
        void f1(){ //Method Overriding
        }
        void f2(int x){ // Method Hiding
        cout<<"Inside B";
        }
};
int main(){
    B obj;
//  obj.f2(); //error'
    obj.f2(4);// B
return 0;
}
```

If the compiler knows at the compile-time which function is called, it is called early binding. If a compiler does not know at compile-time which functions to call up until the run-time, it is called late binding.

## Operator Overloading

C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can make use of the addition operator (+) for string class to concatenate two strings. We know that the task of this operator is to add two operands. So a single operator '+', when placed between integer operands, adds them and when placed between string operands, concatenates them.

```
// C++ program to demonstrate
// Operator Overloading or
// Compile-Time Polymorphism
#include <iostream>
using namespace std;

class Complex {
private:
    int real, imag;

public:
    Complex(int r = 0, int i = 0)
    {
        real = r;
        imag = i;
    }

    // This is automatically called
    // when '+' is used with between
    // two Complex objects
    Complex operator+(Complex const& obj)
    {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
    void print() { cout << real << " + i" << imag << endl; }
};

// Driver code
int main()
{
    Complex c1(10, 5), c2(2, 4);

    // An example call to "operator+"
    Complex c3 = c1 + c2;
    c3.print();
}
```

Output

```
12 + i9
```

**Explanation:** In the above example, the operator '+' is overloaded. Usually, this operator is used to add two numbers (integers or floating point numbers), but here the operator is made to perform the addition of two imaginary or complex numbers.

## 2. Runtime Polymorphism

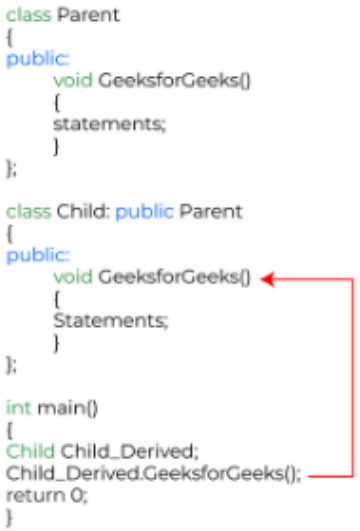
This type of polymorphism is achieved by **Function Overriding**. Late binding and dynamic polymorphism are other names for runtime polymorphism. The function call is resolved at runtime in runtime polymorphism. In contrast, with compile time polymorphism, the compiler determines which function call to bind to the object after deducing it at runtime.

When we call the function using an object of the derived class, the function of the derived class is executed instead of the one in the base class.

So, different functions are executed depending on the object calling the function.

This is known as **function overriding** in C++

Function Overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.




*Function overriding Explanation*

**Runtime Polymorphism with Data Members**

Runtime Polymorphism cannot be achieved by data members in C++. Let's see an example where we are accessing the field by reference variable of parent class which refers to the instance of the derived class.

```cpp
// C++ program for function overriding with data members
#include <bits/stdc++.h>
using namespace std;

//  base class declaration.
class Animal {
public:
    string color = "Black";
};

// inheriting Animal class.
class Dog : public Animal {
public:
    string color = "Grey";
};

// Driver code
int main(void)
{
    Animal d = Dog(); // accessing the field by reference
                      // variable which refers to derived
    cout << d.color;
}
```

**Output**

```
Black
```

We can see that the parent class reference will always refer to the data member of the parent class.

## Virtual Function:

A virtual function in C++ is a base class member function that you can redefine in a derived class to achieve polymorphism. You can declare the function in the base class using the virtual keyword.

A virtual function is a member function that is declared in the base class using the keyword virtual and is re-defined (Overridden) in the derived class.

Some Key Points About Virtual Functions:

- Virtual functions are Dynamic in nature.

- They are defined by inserting the keyword "**virtual**" inside a base class and are always declared with a base class and overridden in a child class
- A virtual function is called during Runtime

```cpp
// C++ Program to demonstrate
// the Virtual Function
#include <iostream>
using namespace std;

// Declaring a Base class
class GFG_Base {

public:
    // virtual function
    virtual void display()
    {
        cout << "Called virtual Base Class function"
             << "\n\n";
    }

    void print()
    {
        cout << "Called GFG_Base print function"
             << "\n\n";
    }
};

// Declaring a Child Class
class GFG_Child : public GFG_Base {

public:
    void display()
    {
        cout << "Called GFG_Child Display Function"
             << "\n\n";
    }

    void print()
    {
        cout << "Called GFG_Child print Function"
             << "\n\n";
    }
};

int main()
{
    // Create a reference of class GFG_Base
    GFG_Base* base;

    GFG_Child child;

    base = &child;

    // This will call the virtual function
    base->display();

    // This will call the non-virtual function
    base->print();
}
```

Output

```
Called GFG_Child Display Function

Called GFG_Base print function
```

```cpp
#include<iostream>
using namespace std;
class A{
    public:
        virtual void f1(){
            cout<<"parent";
        }
};
class B:public A{
    public:
        void f1(){
            cout<<"child";
        }
        void f2(){
        }
};
int main(){
    A *p,objA;
    B objB;
    p=&objB;
    p->f1();
    return 0;
}
```

- **Create a base class named "Shape" and derive two child classes named "Circle" & "Rectangle", create functions: area() & Display() with same signature in each class (function overriding), and the required attributes.**
- **In main function create obj of derived classes and call their respective functions function.**

both task were combined it was required to make virtual functions inside the base class and then override it inside the child classes (rectangle and circle);

- **Create same program but using virtual function and call the override function using pointer of base class.**

there was also a task 2 :
:Create Base class employee with constructor virtual float calculatepay().create derived Class Salaried Employee (fixed salary) Hourly Employee(rate per hour and hours worked) and Comission Employee(ba salary.+percentage of sales). Use RUNTIME polymorphism by storing different employee objects in bas class using pointer. Demonstrate calling calculatepay()through the base class pointer.

**Lab #09 Marks distribution**

| | ER1 | ER6 | ER8 |
|---|---|---|---|
| **Task** | 3points | 3 points | 4 points |

**Lab # 09 Rubric Evaluation Guideline:**

| # | Qualities & Criteria | 0 < Poor <=0.5 | 0.5 < Satisfactory <= 1.5 | 1.5 < Excellent <=2 |
|---|---|---|---|---|
| **ER1** | **Task Completion** | Minimal or no program functionality was achieved. | Some tasks were completed, but the program has errors or incomplete functionalities. | All tasks were completed, and the program runs without errors. |
| # | **Qualities & Criteria** | **0 < Poor <=0.5** | **0.5 < Satisfactory <= 1.5** | **1.5 < Excellent <=2** |
| **ER6** | Program Output | Output is inaccurate or poorly presented. | Output is mostly accurate but may lack labels, captions, or formatting. | Output is clear, accurate, and well presented with labels, captions, and proper formatting. |
| # | **Qualities & Criteria** | **0 < Poor <= 1.5** | **1.5< Satisfactory <= 3** | **3< Excellent <= 4** |
| **ER8** | Question & Answer | Answers some questions but not confidently or based on lab task knowledge. | Answers most questions confidently and based on lab task knowledge. | Answers all questions confidently and demonstrates a deep understanding of the given lab task. |