**National University of Computer & Emerging Sciences, Karachi**

**Computer Science Department**

**Spring 2023, Lab Manual - 10**

|  |  |
| --- | --- |
| **Course Code: CL1004** | **Course : Object Oriented Programming Lab** |
| **Instructor(s) :** |  |

# Contents:

1. Introduction to Polymorphism
2. Types of Polymorphism
   1. Compile time Polymorphism
      1. Function Overloading
   2. Run time Polymorphism
      1. Function Overriding
3. Types of Binding
   1. Early Binding
   2. Late Binding
4. Lab Tasks

# INTRODUCTION TO POLYMORPHISM

The word polymorphism means having many forms.

* Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.
* C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

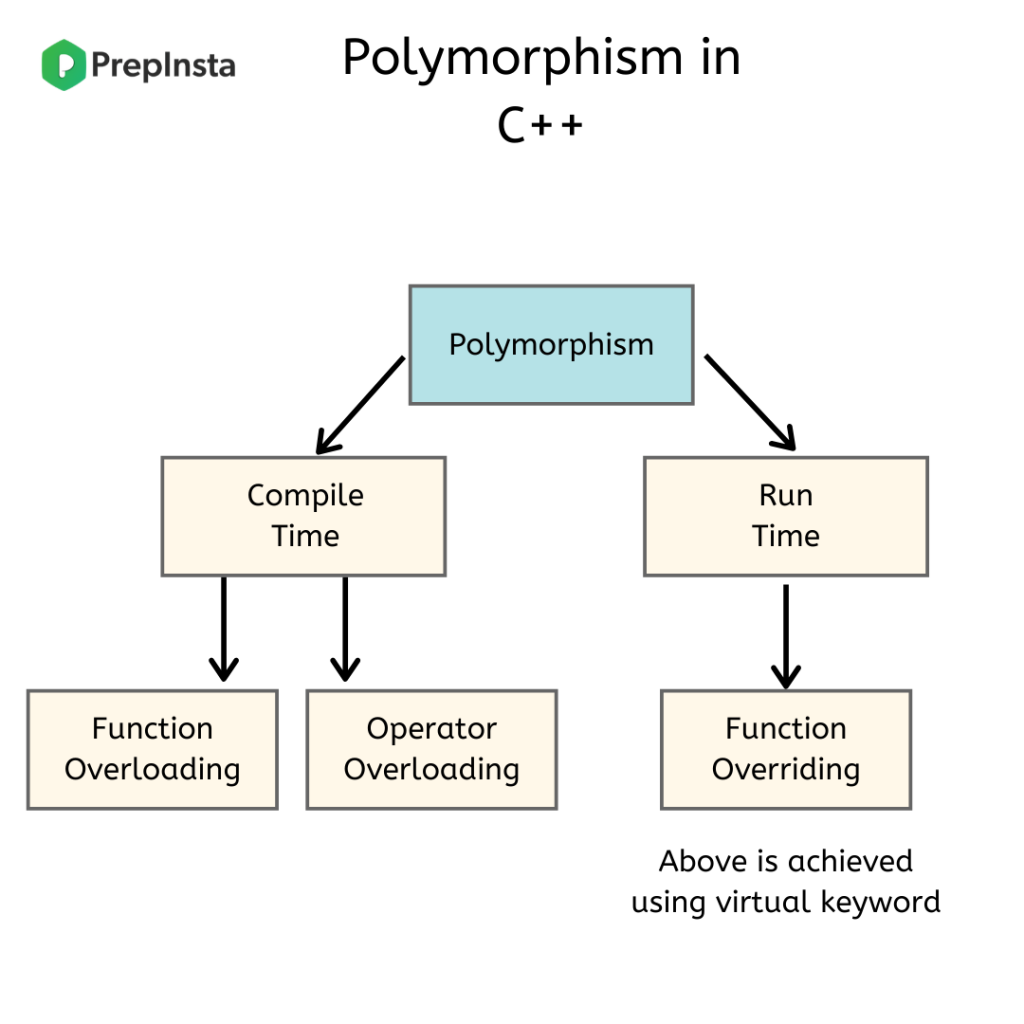
## **Real World Example**:

* A real – life example of polymorphism is that a person at the same time can have different characteristics. A man at the same time is a father, a husband, an employee, so the same person possesses different behavior in different situations. This is called as polymorphism.
* Polymorphism is considered as one of the important features of Object Oriented Programming.

# TYPES OF POLYMORPHISM:

In C++ polymorphism is mainly divided into two types:

* Compile time Polymorphism
* Runtime Polymorphism



## **Compile time Polymorphism:**

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

## **Function Overloading:**

* When there are multiple functions with same name but different parameters then these functions are said to be overloaded.
* Functions can be overloaded by a change in the number of arguments or/and change in the type of arguments.

## **Example Code for Function Overloading**:

**Example 1:**

## 

// C++ program for function overloading

#include <iostream>

using namespace std;

// Function to add two integers

int add(int a, int b) {

return a + b;

}

// Function to add three integers

int add(int a, int b, int c) {

return a + b + c;

}

int main() {

cout << add(1, 2) << endl; // Output: 3

cout << add(1, 2, 3) << endl; // Output: 6

return 0;

}

## **Example 2:**

#include<iostream>

using namespace std;

int mul(int,int);

float mul(float,int);

int mul(int a,int b)

{

return a\*b;

}

float mul(double x, int y)

{

return x\*y;

}

int main()

{

int r1 = mul(6,7);

float r2 = mul(0.2,3);

std::cout << "r1 is : " <<r1<< std::endl;

std::cout <<"r2 is : " <<r2<< std::endl;

return 0;

}

## **Run time Polymorphism:**

This type of polymorphism is achieved by Function Overriding.

## **Function Overriding:**

Function overriding is a feature that allows us to have a same function in child class which is already present in the parent class.

* A child class inherits the data members and member functions of parent class, but when you want to override a functionality in the child class then you can use function overriding. It is like creating a new version of an old function, in the child class.
* To override a function you must have the same signature in the child class.

## **Syntax for Function Overriding:**

public class Parent{

access\_modifier:

return\_type method\_name(){}

};

}

public class child : public Parent {

access\_modifier:

return\_type method\_name(){}

};

}

## **Example Code for Function Overriding:**

#include <iostream>

using namespace std;

class BaseClass {

public:

void disp(){

cout<<"Function of Parent Class";

}

};

class DerivedClass: public BaseClass{

public:

void disp() {

cout<<"Function of Child Class";

}

};

int main() {

DerivedClass obj = DerivedClass();

obj.disp();

return 0;

}

**Sample Run:**

Function of Child Class

**Note: In function overriding, the function in parent class is called the overridden function and function in child class is called overriding function.**

## **Example 2:**

#include <iostream>

using namespace std;

class Animal {

public:

void eat(){

cout<<"Eating...";

}

};

class Dog: public Animal

{

public:

void eat()

{

cout<<"Eating bread...";

}

};

int main(void) {

Dog d = Dog();

d.eat();

return 0;

}

**Output:**

Eating bread...

1. **Types of Binding**

Binding in C++ refers to the process of connecting a function call to its corresponding function implementation. There are two types of binding in C++: early binding (static binding) and late binding (dynamic binding).

Early binding, also known as static binding, occurs at compile-time. When a function call is made, the compiler links the call to the corresponding function implementation based on the declared type of the variable or object. This results in faster program execution because the binding is done during compilation rather than at runtime.

Late binding, also known as dynamic binding, occurs at runtime. When a function call is made, the actual type of the object being pointed to is used to link the call to the corresponding function implementation. This allows for greater flexibility and polymorphism in object-oriented programming.

In C++, late binding is achieved using virtual functions and the virtual keyword. When a function is declared as virtual, the compiler generates a virtual function table (vtable) for the class, which contains pointers to the corresponding function implementations. When a function is called using a pointer to a base class object, the runtime linker uses the vtable to resolve the call to the corresponding function implementation in the derived class.

**a. Early Binding**

#include <iostream>

using namespace std;

class A

{

public:

void show()

{

cout << "Base class" << endl;

}

};

class B: public A

{

public:

void show()

{

cout << "Derived Class" << endl;

}

};

int main()

{

A \*a;

B b;

a = &b;

a->show(); // Early binding

return 0;

}

Early binding is also known as static binding or compile-time binding. It refers to the process of linking a function call to its corresponding function implementation at compile-time, before the program is executed. This results in faster execution of the program because the binding is done during compilation rather than at runtime.

**Example 1**

Output:

**Output:**

**Output:**



**Example 2:**

# include <iostream>

using namespace std;

class Shape {

public:

void draw() {

cout << "Drawing Shape" << endl;

}

};

class Circle : public Shape {

public:

void draw() {

cout << "Drawing Circle" << endl;

}

};

int main() {

Shape\* s = new Circle;

s->draw(); // Early binding: calls Circle's draw() function at compile time

delete s;

return 0;}

}

**Output:**

**Output:**



**b. Late Binding**

In late binding function call is resolved during runtime. Therefore, compiler determines the type of object at runtime, and then binds the function call.

**C++ virtual function**

* A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
* It is used to tell the compiler to perform dynamic linkage or late binding on the function.
* There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
* A 'virtual' is a keyword preceding the normal declaration of a function.
* When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

**Rules of Virtual Function**

* Virtual functions must be members of some class.
* Virtual functions cannot be static members.
* They are accessed through object pointers.
* They can be a friend of another class.
* A virtual function must be defined in the base class, even though it is not used.
* The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
* We cannot have a virtual constructor, but we can have a virtual destructor
* Consider the situation when we don't use the virtual keyword.

**Example 1:**

#include <iostream>

using namespace std;

class Shape {

public:

virtual void draw() {

cout << "Drawing Shape" << endl;

}

};

class Circle : public Shape {

public:

void draw() {

cout << "Drawing Circle" << endl;

}

};

int main() {

Shape\* s;

Circle c;

s = &c;

s->draw(); // Late binding: calls Circle's draw() function at runtime

return 0;

}

**Output:**



**Example 2:**

#include <iostream>

using namespace std;

class A

{

public:

virtual void show()

{

cout << "Base class" << endl;

}

};

class B: public A

{

public:

void show()

{

cout << "Derived Class" << endl;

}

};

int main()

{

A \*a;

B b;

a = &b;

a->show(); // Early binding

return 0;

}

**Output:**



# Exercises:

# Create a Message class with a constructor that takes a single string with a default value. Create a private member string, and in the constructor simply assign the argument string to your internal string. Create two overloaded member functions called print( ): one that takes no arguments and simply prints the message stored in the object, and one that takes a string argument, which it prints in addition to the internal message.

# Write a program to find the area of a rectangle by passing length and breadth as arguments after creating member function in both derived and base class with the same name by technique of function overriding.

# Create a base class area of shape having two parameters (a,b) which can be overloaded in derived class in rectangle and square. Create a class to print the area of a square and a rectangle. The class has two methods with the same name but different number of parameters. The method for printing area of rectangle has two parameters which are length and breadth respectively while the other method for printing area of square has one parameter which is side of square.

### A boy has his money deposited $1000, $1500 and $2000 in banks-Bank A, Bank B and Bank C respectively. We have to print the money deposited by bank. Create a class 'Bank' with a method 'getBalance' which returns 0. Make its three subclasses named 'BankA', 'BankB' and 'BankC' with a method with the same name 'getBalance' which returns the amount deposited in that particular bank. Call the method 'getBalance' by the object of each of the three banks.