National University of Computer & Emerging Sciences, Karachi
Computer Science Department
Spring 2023, Lab Manual - 9

Course Code: CL1004	Course : Object Oriented Programming Lab

Contents:

1. 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.

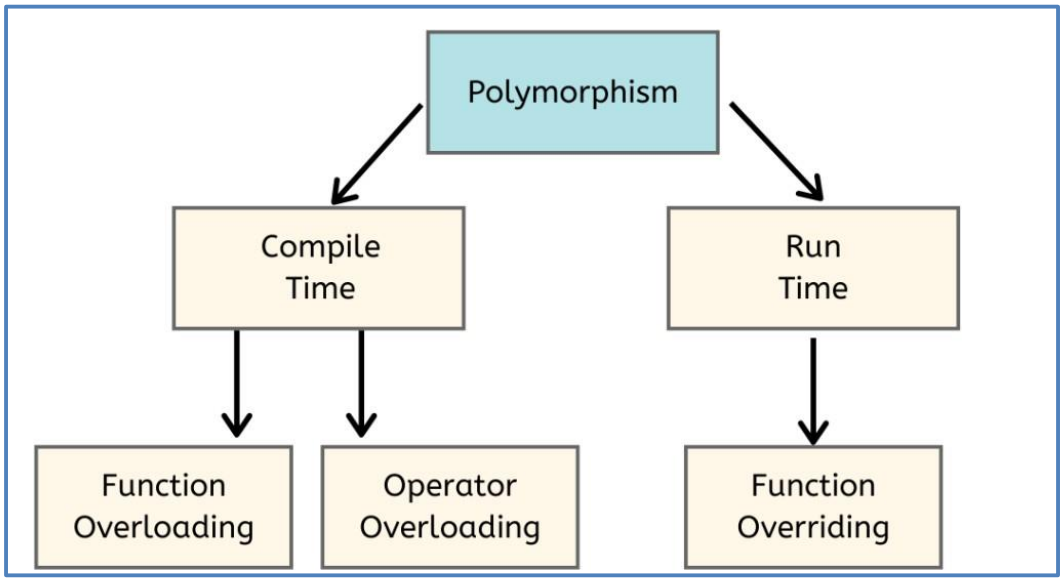
Real World Example:

- 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.

2. TYPES OF POLYMORPHISM:

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

- Compile time Polymorphism
- Runtime Polymorphism



a. Compile time Polymorphism:

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

i. 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;
}
```

b. Run time Polymorphism:

This type of polymorphism is achieved by Function Overriding.

i. 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...
```

3. 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.

```
#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;
```

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:

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;
```

Lab Tasks:

QUESTION 1 :-

You are required to develop a location based System as per the following requirements:

1. Create a class (Location), that takes two inputs latitude (integer) and longitude (integer). The values for both inputs are set through the constructor.
2. The "Location" class is capable of displaying both the values without being able to alter them through Display().
3. Create another class (Details) which extends the functionality of Location class by displaying another location related attribute i.e. address through the function Display().The function being unable to alter the values.
4. Set the value of address through constructor.
5. The "int main()" function should be coded as follows:

a) Create one initialized instance (details) of "Details" class and three initialized instances (obj1, obj2 and obj3) of Location class having the following values (10,20),(5,30) and (90,90) respectively.
b) Display the contents of each instance.
c) Perform the pre increment operation on obj1 and display the result.
d) Perform the post increment operation on obj1 , assign it to obj2 and display the result forobj2.
e) Add "10" to obj1 (keeping "10" the operand on right hand side), assign the result to obj2 and display the result forobj2.
f) Add "10" to obj1 (keeping "10" the operand on left hand side), assign the result to obj2 and display the result forobj2.
g) Assign the contents of obj3 to obj1 and obj2 in a single statement and display the results for all three instances.
h) Reference the instance of "Details" class by a pointer to the "Location" class. Use this pointer to display the address.
i) Return0.

QUESTION 2:-

- A supermarket chain has asked you to develop an automatic checkout system. All products are identifiable by means of a barcode and the product name. Groceries are either sold in packages or by weight. Packed goods have fixed prices. The price of groceries sold by weight is calculated by multiplying the weight by the current price per kilo.

- Develop the classes needed to represent the products first and organize them hierarchically. The Product class, which contains generic information on all products (barcode, name, etc.), can be used as a base class.

- The Product class contains two data members for storing barcodes and the product name. Define a constructor with parameters for both data members. Add default values for the

parameters to provide a default constructor for the class. In addition to the access methods setCode() and getCode(), also define the methods scanner() and printer(). For test purposes, these methods will simply output product data on screen or read the data of a product from the keyboard.

- The next step involves developing special cases of the Product class. Define two classes derived from Product, PrepackedFood and Fresh- Food. In addition to the product data, the PrepackedFood class should contain the unit price and the FreshFood class should contain a weight and a price per kilo as data members. In both classes define a constructor with parameters providing default-values for all data members. Use both the base and member initializer.

- Define the access methods needed for the new data members.Also redefine the methods scanner() and printer() to take the new data members into consideration.

- Test the various classes in a main function that creates three objects each of the types Product, PrepackedFood and FreshFood. One object of each type is fully initialized in the object definition.

- Use the default constructor to create the other object and test the scanner() method. For the third object, test the get and set methods.

- Display the products on screen for all objects.

QUESTION 3:-

- Create a RestaurantMeal class that holds the name and price of a food item served by a restaurant. Its constructor requires arguments for each field.
- Create a HotelService class that holds the name of the service, the service fee, and the room number to which the service was supplied.
- Its constructor also requires arguments for each field.
- Create a RoomServiceMeal class that inherits from both RestaurantMeal and HotelService. Whenever you create a RoomServiceMeal object, the constructor assigns the string "room service" to the name of the service field, and Rs. 400 is assigned to the service fee inherited from HotelService.
- Include a RoomServiceMeal function that displays all of the fields in a RoomServiceMeal by calling display functions from the two parent classes.
- Additionally, the display function should display the total of the meals plus the room service fee.
- In a main()function, instantiate a RoomServiceMeal object that inherits from both classes. For example, a "steak dinner" costing Rs. 2000 is a "room service" provided to room 1202 for a Rs. 400 fee.