**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) : | Eman Shahid |

**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.

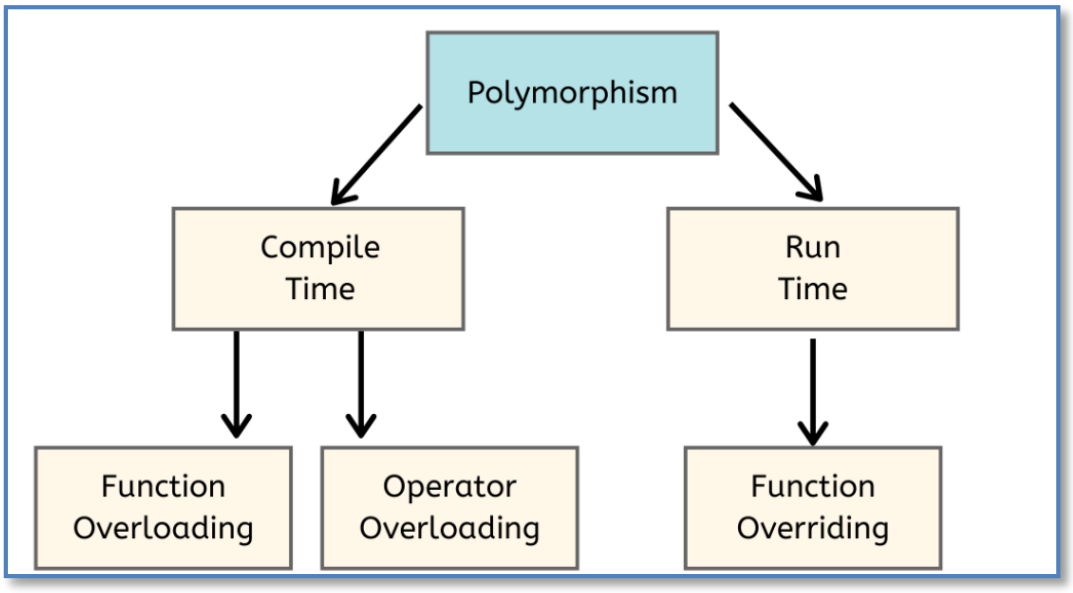
**Real World Example**:

- 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.

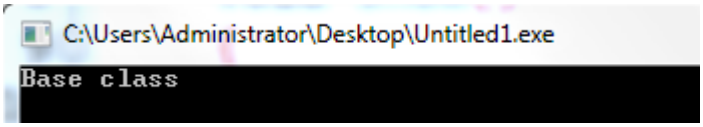
```
#include <iostream>
using namespace std;

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:**

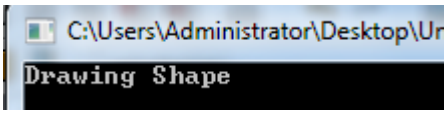
```
#   include <iostream>
using namespace std;

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

**Example 1:**

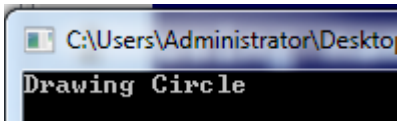
```
#include <iostream>
using namespace std;

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:**

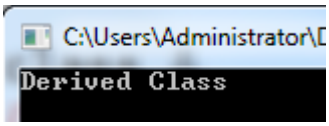
```
#include <iostream>
using namespace std;

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:**



# Lab tasks

**Question # 1**

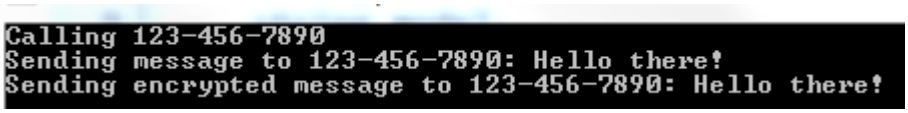
Write a C++ program in which a MobilePhone class with a constructor that takes two parameters, manufacturer and model, and initializes the corresponding member variables. We then define three overloaded member functions: call, sendMessage (with two parameters), and sendMessage (with three parameters).

The call function takes a single parameter phoneNumber and simply prints a message indicating that the phone is calling that number.

The first sendMessage function takes two parameters, phoneNumber and message, and prints a message indicating that the phone is sending a message to that number with the specified message.

The second sendMessage function takes three parameters, phoneNumber, message, and isEncrypted, and checks if the isEncrypted parameter is true. If it is, the function prints a message indicating that the phone is sending an encrypted message to the specified number with the specified message. If not, it calls the first sendMessage function to send a regular message.

In the main function, we create an object of the MobilePhone class with the manufacturer "Nokia" and model "3310". We then test the call and sendMessage functions with different parameters, including a regular message and an encrypted message.




**Question # 2**

Write a C++ program in which a base class RAM and two derived classes, DDR3 and DDR4. The RAM class has two member variables, type and capacity, which store the type and capacity of the RAM. It also has a virtual member function printDetails, which prints the details of the RAM.

The DDR3 class is a derived class of RAM and has a constructor that takes the capacity of the DDR3 RAM as a parameter. It also overrides the printDetails function to print the details of the DDR3 RAM.

The DDR4 class is also a derived class of RAM and has a constructor that takes the capacity of the DDR4 RAM as a parameter. It also overrides the printDetails function to print the details of the DDR4 RAM.

In the main function, we create objects of the DDR3 and DDR4 classes with different capacities, and then call the printDetails function for each object.

**Question # 3**

Write a C++ program in which a class called Mobile represents a mobile phone. The class has two private member variables, brand and model, which store the brand and model of the mobile phone.

Mobile class has a constructor that takes the brand and model as parameters, and two getter functions that return the brand and model.

The interesting part of this program is the operator overloading. Overload the == operator to compare two Mobile objects. In this case, consider two mobile phones to be the same if they have the same brand and model. Define the overloaded == operator as a member function of the Mobile class that takes another Mobile object as a parameter.

In the main function, create three Mobile objects, two of which have the same brand and model. Then compare these objects using the overloaded == operator.

**Question # 4**

Write a C++ program in which a base class base class Compiler and two derived classes, CCompiler and CPPCompiler, which represent C and C++ compilers, respectively. Each of these derived classes overrides the compile function to provide its own implementation of the compiler.

Also define a class TurboCompiler, which derives from both CCompiler and CPPCompiler. This class provides its own implementation of the compile function, which calls both the CCompiler::compile and CPPCompiler::compile functions.

This program also demonstrates the concept of the diamond problem, which arises when a class inherits from two or more classes that in turn inherit from a common base class. In this case, TurboCompiler inherits from both CCompiler and CPPCompiler, which both inherit from Compiler.

the main function begins by creating an object of the TurboCompiler class named turbo.Then, call the compile function on turbo three times. The first two calls use the scope resolution operator :: to explicitly call the compile function of the CCompiler and CPPCompiler classes, respectively. The third call to compile calls the compile function of the TurboCompiler class, which in turn calls the compile functions of both CCompiler and CPPCompile

**Question # 5**

Write a C++ program in which a base class Printer and two derived classes, InkjetPrinter and LaserPrinter, which represent inkjet and laser printers, respectively. Each of these derived classes overrides the print function to provide its own implementation of printing.

In the main function, create two objects of the InkjetPrinter and LaserPrinter classes, respectively. Then, create a pointer to the Printer class named printer.

then set the printer pointer to point to the inkjet object and call the print function on printer. Since printer is a pointer to the base class Printer, late binding is used to determine which version of the print function to call based on the actual object being pointed to at runtime.

Similarly, set the printer pointer to point to the laser object and call the print function on printer. Again, late binding is used to determine which version of the print function to call based on the actual object being pointed to at runtime.