National University of Computer & Emerging Sciences, Karachi
Computer Science Department
Fall 2022, Lab Manual - 08

Course Code: CL-1004	Course: Object Oriented Programming Lab

Lab # 08

Outline:

1. Introduction to Polymorphism
2. Types of Polymorphism
3. Friend Functions
4. Friend Classes
5. Advantages of Friend Functions
6. Disadvantages of Friend Functions
7. 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.

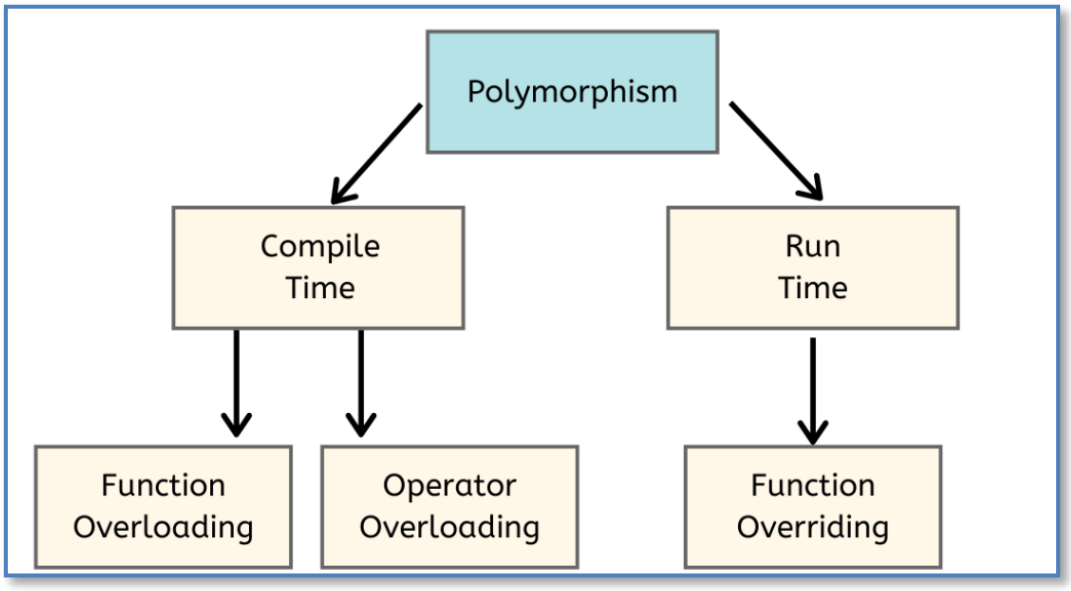
Real World Example:

- 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:

```
// C++ program for function overloading
#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;
   }
};

int main() {

   Geeks obj1;

   // Which function is called will depend on the parameters passed
   // The first 'func' is called
```

```
    obj1.func(7);

    // The second 'func' is called
    obj1.func(9.132);

    // The third 'func' is called
    obj1.func(85,64);
    return 0;
}
```

Sample Run:

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

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

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.

INTRODUCTION TO FRIEND FUNCTIONS:

In object-oriented programming, we have feature of hiding the data through access specifiers. Two of those access specifiers are private and protected. Private members cannot be accessed from outside of class while protected members can only be accessed through derived classes and cannot be accessed from outside of the class. Let's have an example:

```
class MyClass {
   private:
      int member1;
};

int main() {
   MyClass obj;

   // Error! Cannot access private members from here.
   obj.member1 = 5;
}
```

In C++, friend functions are exception to this rule and these functions allow us to access member functions from outside of the class as well. The friend declaration can be placed anywhere in the class declaration.

Friend Functions in C++:

A **friend function** can access the **private** and **protected** data of a class. We declare a friend function using the friend keyword inside the body of the class.

```
class className {
   ... .. ...
   friend returnType functionName(arguments);
   ... .. ...
}
```

Example Code for Friend Functions:

```
// C++ program to demonstrate the working of friend function

#include <iostream>
using namespace std;

class Distance {
   private:
      int meter;

      // friend function
      friend int addFive(Distance);

   public:
      Distance()
     {meter = 0;}

};

// friend function definition
int addFive(Distance d) {

   //accessing private members from the friend function
   d.meter += 5;
   return d.meter;
}

int main() {
   Distance D;
   cout << "Distance: " << addFive(D);
   return 0;
}
```

Sample Run:

```
Distance: 5
```

Example Code for Friend Functions:

```
// Add members of two different classes using friend functions

#include <iostream>
using namespace std;

// forward declaration
class ClassB;

class ClassA {
private:
    int numA;

public:
    // constructor to initialize numA to 12
    ClassA() : numA(12) {}

   // friend function declaration
     friend int add(ClassA, ClassB);
};
class ClassB {
  public:
    // constructor to initialize numB to 1
    ClassB() : numB(1) {}

  private:
    int numB;

    // friend function declaration
    friend int add(ClassA, ClassB);
};

// access members of both classes
int add(ClassA objectA, ClassB objectB) {
  return (objectA.numA + objectB.numB);
}

int main() {
  ClassA objectA;
  ClassB objectB;
  cout << "Sum: " << add(objectA, objectB);
  return 0;
}
```

Sample Run:

```
Sum: 13
```

In this program, ClassA and ClassB have declared **add()** as a friend function. Thus, this function can access **private** data of both classes. One thing to notice here is the friend function inside ClassA is using the ClassB. However, we haven't defined ClassB at this point.

```
// inside classA
friend int add(ClassA, ClassB);
```

For this to work, we need a forward declaration of ClassB in our program:

```
// forward declaration
class ClassB;
```

Friend Classes in C++:

```
class ClassB;

class ClassA {
  // ClassB is a friend class of ClassA
  friend class ClassB;
  ... .. ...
}

class ClassB {
  ... .. ...
}
```

When a class is declared a friend class, all the member functions of the friend class become friend functions.

Since ClassB is a friend class, we can access all members of ClassA from inside ClassB.

However, we cannot access members of ClassB from inside ClassA.

Example Code for Friend Classes:

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

class Square;

class Rectangle {
	int width, height;
public:
	Rectangle(int w, int h){
		width=w; height=h;
		}
	void display() {
		cout << "Rectangle: " << width * height << endl;
	};
	void morph(Square &);
};

class Square {
	int side;
public:
	Square(int s)
	{side=s;}
	void display() {
		cout << "Square: " << side * side << endl;
	};
	friend class Rectangle;
};

void Rectangle::morph(Square &s) {
	width = s.side;
	height = s.side;
}

int main () {
	Rectangle rec(5,10);
	Square sq(5);
	cout << "Before:" << endl;
	rec.display();
	sq.display();
	rec.morph(sq);
	cout << "\nAfter:" << endl;
	rec.display();
	sq.display();
	return 0;
}
```

Sample Run:

```
Before:
Rectangle: 50
Square: 25

After:
Rectangle: 25
Square: 25
```

We declared **Rectangle** as a friend of **Square** so that **Rectangle** member functions could have access to the private member, **Square::side** In our example, **Rectangle** is considered as a friend class by **Square** but **Rectangle** does not consider **Square** to be a friend, so **Rectangle** can access the private members of **Square** but not the other way around.

Advantages/Applications of Friend Functions:

- Windows will have many functions that should not be publicly accessible but that can be needed by a related class like WindowsManager so friend functions can be used.

- You want to operate on objects of two different classes then friend function can be useful as we observed in above examples otherwise without friend function in this situation, your code will be long, complex and hard.

Disadvantages of Friend Functions:

- One of the distinguishing features of C++ is encapsulation i.e., bundling of data and functions operating on that data together so that no outside function or class can access the data. But by allowing the friend functions or class to access the private members of another class, we actually compromise the encapsulation feature.

- In order to prevent this, we should be careful about using friend functions or class. We should ensure that we should not use too many friend functions and classes in our program which will totally compromise on the encapsulation.

LAB TASKS:

Task - 01:

Create a class called **Calculator** that has three private member variables:

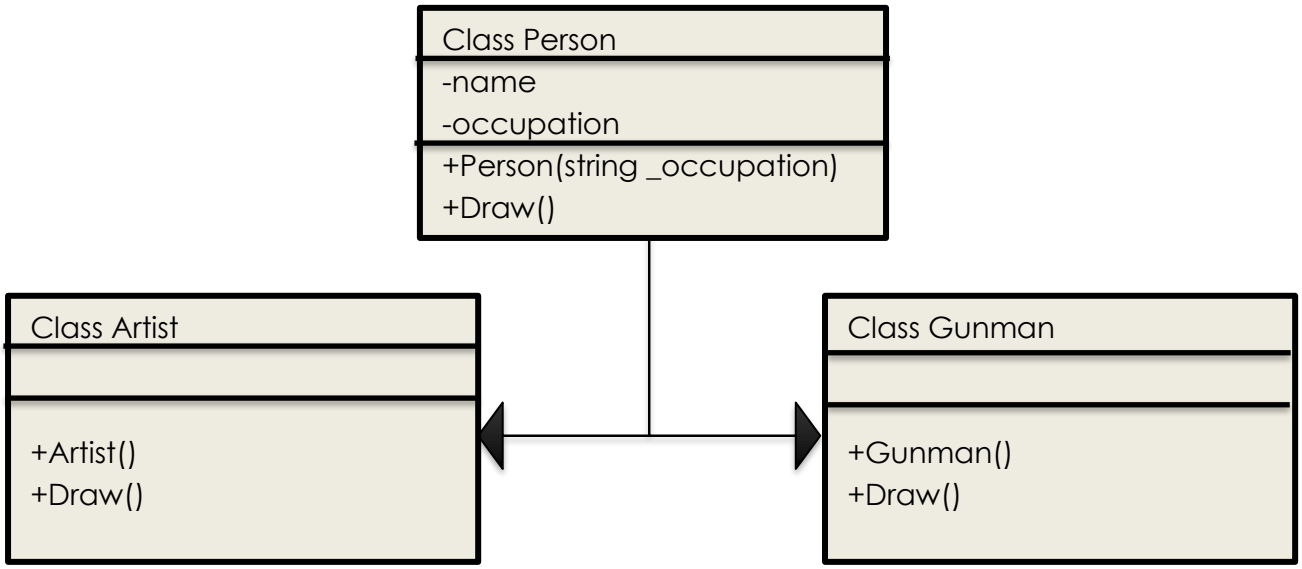
- Num1
- Num2
- Num3

In this class, you have to overload the functions for **addition** and **multiplication** such that they take two and three inputs respectively. You also have to make **methods** for subtraction and **division**.

For example: add(1,2) and add(1,2,3) similarly for multiply, it would be multiply(1,2) and multiply(1,2,3)

You may ask the user for input at the time of object creation. Afterwards just demonstrate how the functions are being called.

Task - 02:



Create the classes following the diagram shown. Keep the following things in mind:

- When an object of Artist is created, the value "artist" will be set to occupation.
- When an object of Gunman is created, the value "gunman" will be set to occupation.
- Person::Draw() will print out "A person can draw in many ways"
- Artist::Draw() will print out "An artist can draw with a paint brush"
- Gunman::Draw() will print out "A gunman draws a gun to shoot"

Task - 03:

Create a class named **LoneClass** with the following private members:

- A string called **loneObjName**
- An integer called **loneObjID**

Create a **friend function** in the class by the name of **friendOfLoneClass** which takes an object of a lone class as parameter.
The friend function should display the following: "I am friend of this Lone Class with ObjName = <object_name> and ObjID = <object_ID>"

Task - 04:

Create a class called **ShapeDetails** with these private attributes:

- Area
- Parameter

Do not make any getters and setter for these, as we will be observing how friend classes will be used.
Create two classes called **Square** and **Circle** which will be friends of the class.
Class **Square** will have the following attributes:

- Side_length

Class **Circle** will have the following attributes:

- Radius_length

Both classes will have these two functions that take an object of **ShapeDetails** as input:

- Calculate_Perimeter
- Calculate_Area

In you main program you will have to make two objects for **ShapeDetails** and one object each of **Square** and **Circle**
Your task is to demonstrate how you will store the values of perimeter and area in the **ShapeDetails** object by using objects of **Square** and **Circles**