National University of Computer & Emerging Sciences, Karachi
Computer Science Department
Spring 2022, Lab Manual - 07

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

Lab # 07

Outline:

1. Types of Inheritance based on Derived Classes (Multiple and Hybrid)
2. Constructors in Inheritance
3. UML Representation for Inheritance
4. Lab Tasks

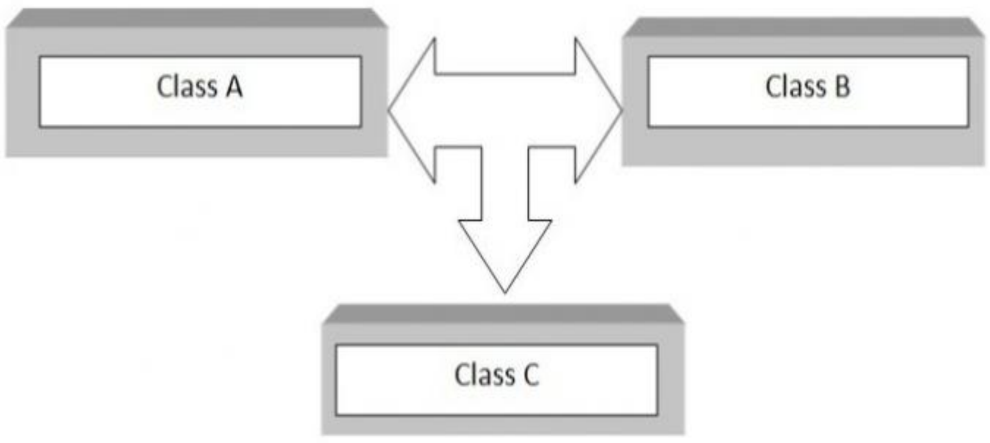
TYPES OF INHERITANCE BASED ON DERIVED CLASSES

Inheritance based on derived classes can be categorized as follows:

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Multiple Inheritance:

- In multiple inheritance, a class is derived from two or more base classes. In multiple inheritance a derived class has more than one base class.
- As shown in the figure below, class C is derived from two base classes A and B.



Syntax for multiple Inheritance:

```
class A   // base class
{
  // body of the class
};
class B  // base class
{
  // body of the class
};
class C : acess_specifier A,  acess_specifier B     // derived class
{
    // body of the class
};
```

Example code for multiple Inheritance:

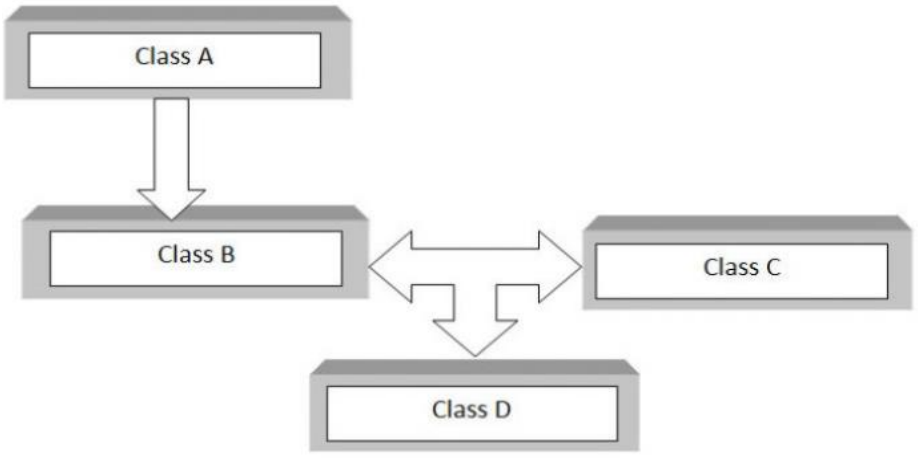
```
#include
using namespace std;
class A // base class
{
     public:
     int x;
     void getx()
    {
         cout << "enter value of x: "; cin >> x;
    }
};
class B // base class
{
     public:
     int y;
     void gety()
     {
         cout << "enter value of y: "; cin >> y;
     }
};
class C : public A, public B   //C is derived from class A and
class B
{
     public:
     void sum()
     {
         cout << "Sum = " << x + y;
     }
};
int main()
{
      C obj1; //object of derived class C
      obj1.getx();
      obj1.gety();
      obj1.sum();
      return 0;
}    //end of program
```

Sample Run

```
enter value of x: 5
enter value of y: 4
Sum = 9
```

Hybrid Inheritance:

- The inheritance in which the derivation of a class involves more than one form of any inheritance is called hybrid inheritance.
- Basically C++ hybrid inheritance is combination of two or more types of inheritance. It can also be called multi path inheritance.
- The figure below shows the hybrid combination of single inheritance and multiple inheritance. Hybrid inheritance is used in a situation where we need to apply more than one inheritance in a program.



Syntax for hybrid Inheritance:

```
class A   // base class
{
  // body of the class
};
class B : public A
{
  // body of the class
};
class C
{
    // body of the class
};
class D : public B, public C
{
    // body of the class
};
```

Example code for hybrid Inheritance:

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

class A
{
    public:
    int x;
};
class B : public A
{
    public:
    B()     //constructor to initialize x in base class A
    {
      x = 10;
    }
};
class C
 {
    public:
    int y;
    C()   //constructor to initialize y
    {
       y = 4;
     }
};
class D : public B, public C   //D is derived from class B and class C
{
    public:
    void sum()
    {
       cout << "Sum= " << x + y;
    }
};
int main()
{
     D obj1;         //object of derived class D
    obj1.sum();
    return 0;
}           //end of program
```

Sample Run

```
Sum= 14
```

CONSTRUCTORS IN INHERITANCE

Constructors are a special type of member function, which are automatically invoked when you create an object of a class.
When an object of a class is created, the constructors are called in the following order:

1. Constructor for the **Parent** class
2. Constructor for the **Child** class

Let us observe the order in which the constructors are being called in the following code:

```
class Parent {
public:
        Parent() { //constructor for parent class
                cout << "Parent Constructor\n";
        }
};

class Child :public Parent{
public:
    Child() { //constructor for child class
                cout << "Child Constructor\n";
        }
};

int main() {

        Child ch;

        return 0;

}
```

Next, if you want to invoke a parametrized constructor for parent class, you cannot do this directly. For this you have to use **Initializer Lists**.

Example code for Invoking Constructors for Parent Class:

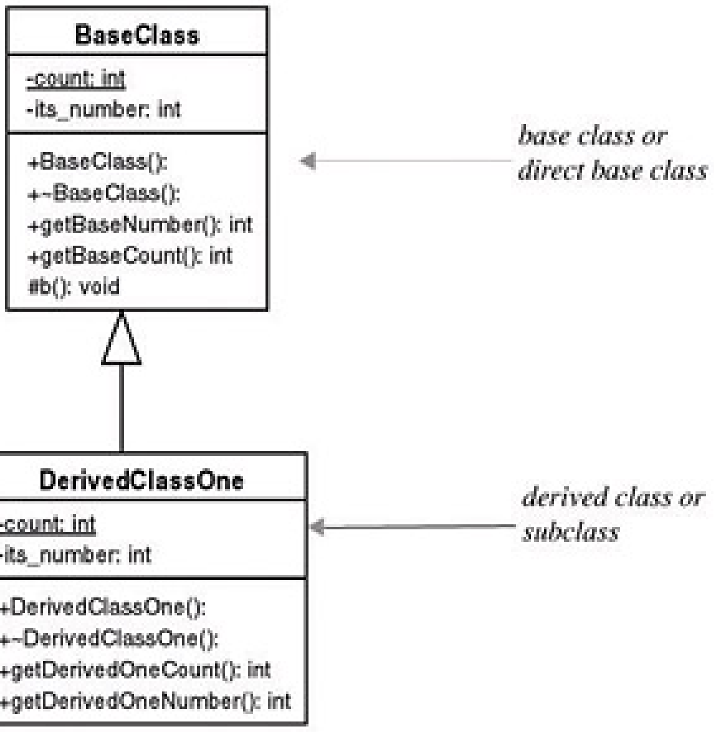
```
#include <iostream>

using namespace std;


class base {
    int a;
    public:
    base() { // constructor for base class
            cout << "Constructor: Base.\n";
    }
    base(int _a) { // parametrized constructor for base class
            cout << "Parametrized Constructor: Base.\n";
            a = _a;
    }
    int get_a() {
            return a;
    }
};

class derived : public base{
    public:
    derived() { //constructor for derived class
            cout << "Constructor: Derived.\n";
    }
    derived(int _a):base(_a) { // parametrized constructor for derived class
            cout << "Parametrized Constructor: Derived.\n";
    }
};

int main() {
    derived ch; //object for derived class
    derived child(20); //object for derived class with value = 20
    cout << "Value of a = " << child.get_a() << endl;
    return 0;
}
```

Sample Run

```
Constructor: Base.
Constructor: Derived.
Parametrized Constructor: Base.
Parametrized Constructor: Derived.
Value of a = 20
```

UML REPRESENTATION FOR INHERITANCE

An association line tipped with an open arrowhead is drawn from the subclass to the base class. The figure below shows a class diagram for two classes named BaseClass and DerivedClassOne.



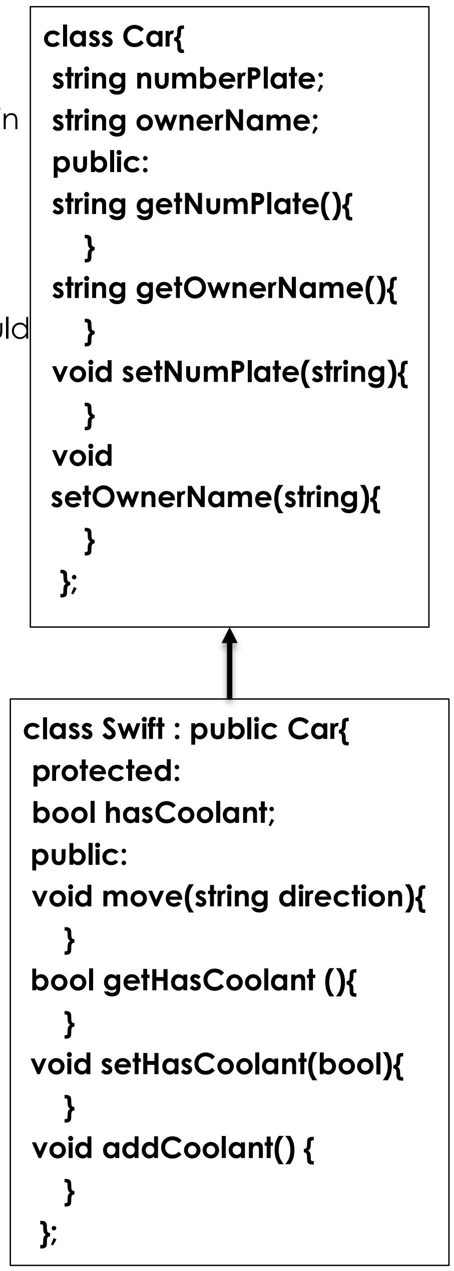
BaseClass is referred to as the base class and DerivedClassOne is referred to as the derived class or subclass.

BaseClass contains two private attributes. One is a static, class-wide variable named count, and the other is an integer variable named its_count. BaseClass contains four public functions and one protected function.

LAB TASKS:

Task - 01:

Given the UML Diagram, create the classes according to it. There will be a class called **Car** which will contain the number plate and the car owner's name. It should also contain accessor and mutator functions for the data members. Create another class **Swift** which is inherited from the class **Car.** It should have a boolean member called **hasCoolant**. Create accessor an mutator functions for it as well. It should also have a function called **addCoolant()** which should allow user to add coolant to the car if it doesn't have any. It should also update the value of **hasCoolant** after successfully adding the coolant.
You must make use of parametrized constructors to initialize the values.



```
class Car{
 string numberPlate;
 string ownerName;
 public:
 string getNumPlate(){
    }
 string getOwnerName(){
    }
 void setNumPlate(string){
    }
 void
 setOwnerName(string){
    }
  };
```

```
class Swift : public Car{
 protected:
 bool hasCoolant;
 public:
 void move(string direction){
    }
 bool getHasCoolant (){
    }
 void setHasCoolant(bool){
    }
 void addCoolant() {
    }
  };
```

Task - 02:

A storehouse stores multiple items in it. Some of them are: **school books, pencils, color pencils, sharpeners, erasers**. Create a class called **StorehouseItems** that has the following data members:

- **quantityOfBoxes** – number of boxes of markers/colors/sharpeners/erasers
- **quantiyPerBox** – number of markers/colors/sharpeners/erasers in a box

Create one class for each: **SchoolBooks, Pencils, Sharpeners, Erasers** which are all inherited from the **StorehouseItems** class. Create a class for **ColorPensils** which will be inherited from **Pencils.** Create proper accessor and mutators for your classes. Your program should demonstrate creating objects for each of the classes, then setting values for the various quantities for the objects.

Each class should have a function that displays the total amount of markers/colors/sharpeners/erases. (Total amount = quantity of boxes * quantity per box)
You must make use of parametrized constructors to initialize the values.

Task - 03:

Create a class named **Shape.** All our shapes will be inherited from this class. It will contain the following data members and
functions:

- **numberOfSides**
- **area**
- parametrized constructor
- Create accessors and mutators for the data members.

Create classes called **Rectangle, Circle** and
Triangle, which are all inherited from the class **Shape.** Create a class called **Square** which is inherited from **Rectangle**.
The classes will have the following members:

Rectangle:
- **length**
- **width**
- parametrized constructor
- generateArea() – should place the result in area

Circle:
- **radius**
- parametrized constructor
- generateArea() – should place the result in area

Triangle:
- **height**
- **base**
- parametrized constructor
- generateArea() – should place the result in area (Area = height*base/2)

Square:
- It should have a parametrized constructor that takes **one side** as input. The constructor should call the constructor for **Rectangle** class with that value as parameters.
- checkSides(); - checks if both sides are equal. Sides are inherited from **Rectangle**.
- generateArea() – should place the result in area

You must make use of parametrized constructors to initialize the values.