

Lab 10

OBJECTIVE :

Things that will be covered in today's lab:

- Polymorphism
- Abstract classes and pure virtual function

THEORY :

Polymorphism is the term used to describe the process by which different implementations of a function can be accessed via the same name. For this reason, polymorphism is sometimes characterized by the phrase “one interface, multiple methods”.

In C++ polymorphism is supported both run time, and at compile time. Function overloading is an example of compile-time polymorphism. Run-time polymorphism is accomplished by using inheritance and virtual functions.

One of the key features of class inheritance is that a pointer to a derived class is type-compatible with a pointer to its base class. *Polymorphism* is the art of taking advantage of this simple but powerful and versatile feature.

Task1:

Compile and run the code below:

```
class XYZ
{
    public:
    void print()
        { cout<<"Parent class print:"<<endl; }
};
class ABC: public XYZ
{
    public:
    void print() { cout<<"child class print:"<<endl; }
};
void main( )
{
    XYZ *xyz;
    ABC abc;
    xyz = &abc; // store the address of abc
    xyz->print(); // call abc print .
}
```

Did it give the following output?

Parent class print:

The reason for the incorrect output is that the call of the function *print()* is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed. This is also sometimes called **early binding** because the *print()* function is set during the compilation of the program. But now, let's make the following slight modification in our program:

```
class XYZ
{
    public:
    virtual void print()
        { cout<<"Parent class print:"<<endl; }
};
```

Compile and run the code above and observe the output which would be

```
child class print :
```

This time, the compiler looks at the contents of the pointer instead of its type. Hence, since address of object of “ABC” class is stored in *xyz the respective *print()* function is called. As you can see, each of the child classes has a separate implementation for the function *print()*. This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

Virtual Function:

A virtual function is a function in a base class that is declared using the keyword *virtual*.

Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**.

Task2:

Compile and run the code below:

```
class Animal
{
    public:
    virtual void speak() { cout<<"Animal speaks\n"; }
};
class Cat: public Animal
{
    public:
    void speak() { cout<<"Mew mew\n"; }
};
class Dog: public Animal
{
    public:
    void speak() { cout<<"woof woof\n"; }
```

```

};
Class Cow: public Animal
{
    public:
    void speak() { cout<<"Moo moo\n"; }
};

int main()
{
    Animal *pAnim;
    Animal anim;
    Cat mano;
    Dog qatmeer;
    Cow moti;
    pAnim = &anim;
    pAnim ->speak();

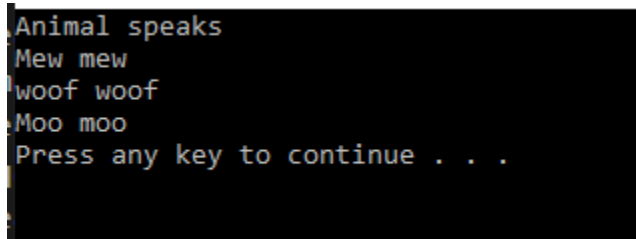
    pAnim = &mano;
    pAnim->speak();
    pAnim =&qatmeer;
    pAnim->speak();

    pAnim =&moti;
    pAnim->speak();

    return 0;
}

```

Observe and write the output in the space below:



```

Animal speaks
Mew mew
woof woof
Moo moo
Press any key to continue . . .

```

Abstract classes and Pure virtual function

- ▶ When we never want to instantiate objects of a base class, we call it an *abstract class*. Such a class exists only to act as a parent of derived classes that will be used to instantiate objects. It may also provide an interface for the class hierarchy.
- ▶ We can restrict a user from creating an object of base class by adding at least one pure virtual function

- ▶ A pure virtual function is one with the expression =0 added to the declaration

Task 3:

Please make the following change in the Animal class (of task2 above) to make it an abstract class and compile the code

```
class Animal
{
    public:
    virtual void speak()=0
};
```

Did the following line in the main function error out?

```
Animal anim; Ans: Yes
```

This is because a class with a pure virtual function is an abstract class and you can't instantiate objects from it (although you might from classes derived from it).

- ▶ Once you've placed a pure virtual function in the base class, you must override it in all the derived classes from which you want to instantiate objects.
- ▶ If a class doesn't override the pure virtual function, it becomes an abstract class itself. For consistency, you may want to make all the virtual functions in the base class pure.

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

class Animal {
public:
    virtual void speak() = 0;
};

class Dog : public Animal {
public:
    void speak() {
```

```

        cout << "Woof!" << endl;
    }
};

class Cat : public Animal {
public:
    void speak() {
        cout << "Meow!" << endl;
    }
};

class Cow :public Animal {
public:
    void speak() { cout << "Moo moo\n"; }
};

int main() {
    // Animal anim; // This line will error out since Animal is an abstract class
    Animal* pAnim;
    Cat mano;
    Dog qatmeer;
    Cow C;

    pAnim = &mano;
    pAnim->speak();

    pAnim = &qatmeer;
    pAnim->speak();

    pAnim = &C;
    pAnim->speak();

    return 0;
}

```

Output:

```

Meow!
Woof!
Moo moo

D:\Lab 10\x64\Debug\Lab 10.exe (
Press any key to close this wind

```

Task4:

We have to develop a software to manage an animal farm consisting of cats, dogs and cows. Partial code for it is given below and you can reuse the code given for Task3 for Animal/Cat/Dog/Cow classes here.

```
class AnimalFarm
{
    Public:
        AnimalFarm();
        AnimalFarm(int pCap);

        //adds the animal pointed by the Animal pointer it could be cat/dog/cow
        void AddAnimal(Animal *ptr);
        void AnimalsTalk();

        ~AnimalFarm();

    Private:
        Animal **aPtr; //composition relationship with class Animal
        int capacity; //default capacity is 5
        int currentCount; // count of animals default is zero
}

AnimalFarm::AnimalFarm()
{
    currentCount=0;
    capacity=5;
    aPtr= new Animal*[capacity];
}

AnimalFarm::AnimalFarm(int pCap)
{
    Capacity=pCap;
    aPtr= new Animal*[capacity];
    currentCount=0;
}

void AnimalFarm::AddAnimal(Animal *ptr)
{
    if(currentCount<capacity)
    {
        aPtr[currentCount]=ptr;
        currentCount++;
    }
    else
    {
        cout<<"Full capacity can't add another animal";
    }
}
```

```

void AnimalFarm::AnimalsTalk()
{
    for(int i=0; i<currentCount; i++)
    {
        aPtr[i]->Speak();
    }
}
AnimalFarm::~~AnimalFarm()
{
    if(aPtr) delete []aPtr; //deallocate memory
}

int main()
{
    AnimalFarm *Frm= new AnimalFarm();
    Cat *c1= new Cat();
    Cat *c2= new Cat();
    Dog *d = new Dog();
    Cow *cw1= new Cow();
    Cow *cw2= new Cow();
    Frm-> AddAnimal( c1);
    Frm-> AddAnimal( d);
    Frm-> AddAnimal( cw1);
    Frm-> AddAnimal( c2);
    Frm-> AddAnimal( cw2);

    Frm-> AnimalsTalk();
    delete Frm;

    return 0;
}

```

Compile and run your code and observe the output. Make changes to main function to create an animal farm using the parameterized constructor, add animals accordingly. Compile, run and observe the output.

```

#include <iostream>
using namespace std;

class Animal {
public:
    virtual void speak() = 0; // Pure virtual function
};

class Cat : public Animal {
public:
    void speak()
    {
        cout << "Meow" << endl;
    }
};

class Dog : public Animal {
public:

```

```

        void speak()
        {
            cout << "Woof" << endl;
        }
};

class Cow : public Animal {
public:
    void speak() {
        cout << "Moo" << endl;
    }
};

class AnimalFarm {
public:
    AnimalFarm();
    AnimalFarm(int pCap, int currentCount);
    void AddAnimal(Animal *ptr);
    void AnimalsTalk();
    ~AnimalFarm();

private:
    Animal **aPtr;
    int capacity;
    int currentCount;
};

AnimalFarm::AnimalFarm() {
    currentCount = 0;
    capacity = 5;
    aPtr = new Animal*[capacity];
}

AnimalFarm::AnimalFarm(int pCap, int currentCount) {
    capacity = pCap;
    aPtr = new Animal*[capacity];
    currentCount = currentCount;
}

void AnimalFarm::AddAnimal(Animal *ptr) {
    if (currentCount < capacity) {
        aPtr[currentCount] = ptr;
        currentCount++;
    }
    else {
        cout << "Full capacity can't add another animal";
    }
}

void AnimalFarm::AnimalsTalk() {
    for (int i = 0; i < currentCount; i++) {
        aPtr[i]->speak();
    }
}

AnimalFarm::~~AnimalFarm() {
    if (aPtr) delete[] aPtr;
}

```



```

int main() {
    AnimalFarm *Frm = new AnimalFarm(7,4); // Using parameterized constructor
    Cat *c1 = new Cat();
    Cat *c2 = new Cat();
    Dog *d = new Dog();
    Cow *cw1 = new Cow();
    Cow *cw2 = new Cow();

    Frm->AddAnimal(c1);
    Frm->AddAnimal(d);
    Frm->AddAnimal(cw1);
    Frm->AddAnimal(c2);
    Frm->AddAnimal(cw2);

    Frm->AnimalsTalk();

    delete Frm;
    system("pause");
    return 0;
}

```

This is currentcount initially it is zero , So it is zero not seven

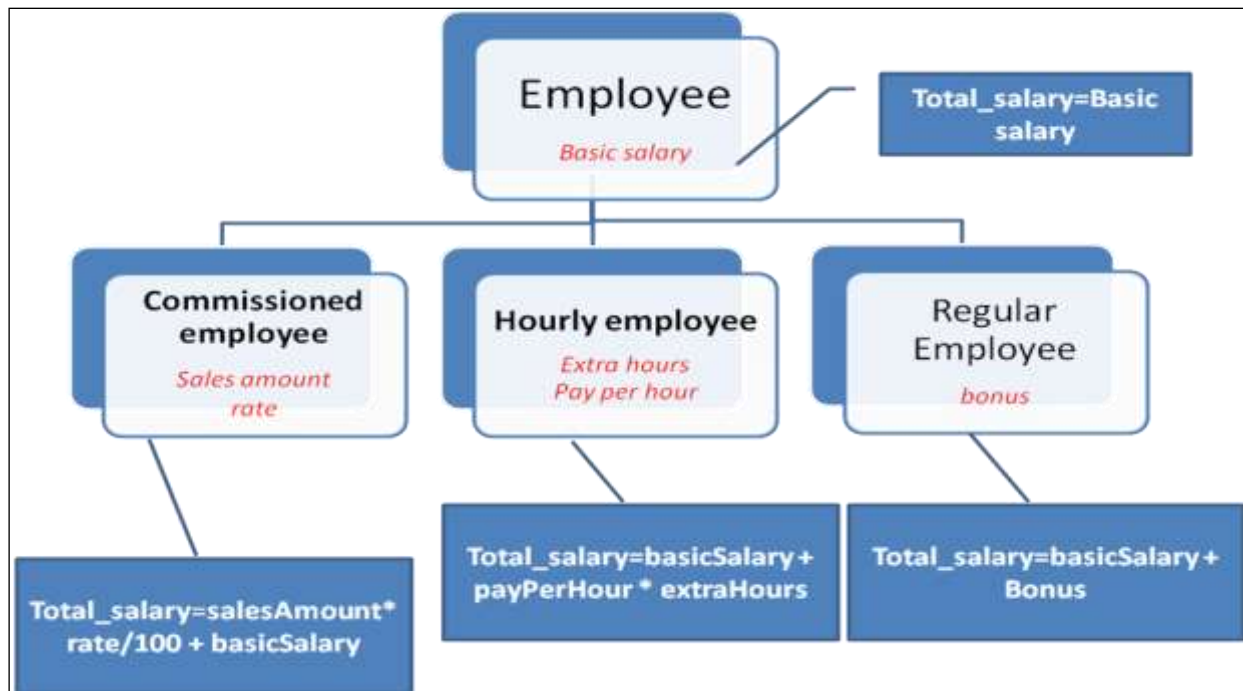
After Changes:

```

Meow
Woof
Moo
Meow
Moo
Press any key to continue . . .

```

Task5: We want to design a system for a company to calculate salaries of different types of employees. Consider the hierarchy shown in Figure below



Every employee has an employee ID and a basic salary. The Commissioned employee has a sales amount and rate. Hourly employee is paid on the basis of number of working hours. A regular employee may have a bonus.

You have to implement all the above classes. Write constructor for all classes. The main functionality is to calculate salary for each employee which is calculated as follows:

Commissioned Employee: $\text{Total Salary} = \text{sales amount} * \text{rate} / 100 + \text{basic salary}$

Hourly Employee: $\text{Total salary} = \text{basic salary} + \text{pay per hour} * \text{extra hours}$

Regular Employee: $\text{Total salary} = \text{basic salary} + \text{bonus}$

You have to define the following function in all classes:

float calculateSalary() and run the given **main()** for the following two cases:

1. when the *calculateSalary()* in base class is not **virtual**
2. when the *calculateSalary()* in base class is made **virtual**

Use the following main().

```

int main()
{
    CommissionedEmployee E1(25, 5000, 1000, 10);
    // CASE 1 - derived Class Pointer pointing to Derived class object
    CommissionedEmployee * ptr;
    ptr = &E1;
}
  
```

```

cout<<" Commissioned Employee salary:"<<ptr->calculateSalary();
cout<<endl;
// CASE 2 - Base Class Pointer pointing to Derived class object
Employee * eptr;
eptr = &E1;
cout<<" Commissioned Employee salary:"<<eptr->calculateSalary();
cout<<endl;
CommissionedEmployee E2 (25, 5000, 1000, 10);
CommissionedEmployee E3 (26, 5000, 2000, 10);
HourlyEmployeeH1(27, 5000, 10, 100 );
HourlyEmployeeH2(28, 5000, 5, 100 );
RegularEmployeeR1(29, 5000, 1000 );
RegularEmployeeR2(29, 5000, 2000 );

Employee * list [6];
list[0] = &E2;
list[1] = &E3;
list[2] = &H1;
list[3] = &H2;
list[4] = &R1;
list[5] = &R2;

for(int i = 0 ; i < 6; i++)
{
cout<<"Employee "<<i<<" salary is : "<<list[i]->calculateSalary();
cout<<endl;
}

return 0;
}

```

```

#include <iostream>
using namespace std;

class Employee {
protected:
    int empId;
    float basicSalary;

public:
    Employee(int id, float salary) : empId(id), basicSalary(salary) {}
    virtual ~Employee() {}

    virtual float calculateSalary() {
        return basicSalary;
    }
}

```

use this
virtual float calculateSalary = 0;

```

};

class CommissionedEmployee : public Employee {
    float salesAmount;
    float rate;

public:
    CommissionedEmployee(int id, float salary, float sales, float commissionRate)
        : Employee(id, salary), salesAmount(sales), rate(commissionRate) {}

    float calculateSalary() {
        return (salesAmount * rate / 100) + basicSalary;
    }
};

class HourlyEmployee : public Employee {
    float payPerHour;
    float extraHours;

public:
    HourlyEmployee(int id, float salary, float payHour, float extraHrs)
        : Employee(id, salary), payPerHour(payHour), extraHours(extraHrs) {}

    float calculateSalary() {
        return basicSalary + (payPerHour * extraHours);
    }
};

class RegularEmployee : public Employee {
    float bonus;

public:
    RegularEmployee(int id, float salary, float bonusAmount)
        : Employee(id, salary), bonus(bonusAmount) {}

    float calculateSalary() {
        return basicSalary + bonus;
    }
};

int main() {
    // CASE 1 - derived Class Pointer pointing to Derived class object
    CommissionedEmployee E1(25, 5000, 1000, 10);
    CommissionedEmployee* ptr;
    ptr = &E1;
}

```

```

    cout << "Commissioned Employee salary: " << ptr->calculateSalary() << endl;

    // CASE 2 - Base Class Pointer pointing to Derived class object
    Employee* eptr;
    eptr = &E1;
    cout << "Commissioned Employee salary: " << eptr->calculateSalary() << endl;

    // Create other employees
    CommissionedEmployee E2(25, 5000, 1000, 10);
    CommissionedEmployee E3(26, 5000, 2000, 10);
    HourlyEmployee H1(27, 5000, 10, 100);
    HourlyEmployee H2(28, 5000, 5, 100);
    RegularEmployee R1(29, 5000, 1000);
    RegularEmployee R2(29, 5000, 2000);

    Employee* list[6];
    list[0] = &E2;
    list[1] = &E3;
    list[2] = &H1;
    list[3] = &H2;
    list[4] = &R1;
    list[5] = &R2;

    for (int i = 0; i < 6; i++) {
        cout << "Employee " << i << " salary is : " << list[i]->calculateSalary()
<< endl;
    }

    return 0;
}

```

Microsoft Visual Studio Debug Console

```

Commissioned Employee salary: 5100
Commissioned Employee salary: 5100
Employee 0 salary is : 5100
Employee 1 salary is : 5200
Employee 2 salary is : 6000
Employee 3 salary is : 5500
Employee 4 salary is : 6000
Employee 5 salary is : 7000

D:\Lab 10\x64\Debug\Lab 10.exe (process 21424) exited with code 0.
Press any key to close this window . . .

```