

Lab 10
Polymorphism

OBJECTIVE:

Things that will be covered in today's lab:

- Polymorphism

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. Operator and function overloading are examples 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.

```
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 .
}
```

When the above code is compiled and executed, it produces the following result:

```
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 a slight modification in our program and precede the declaration of `print()` in the "abc" class with the keyword **virtual**. After this slight modification, when the previous example code is compiled and executed, it produces the following result:

```
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**.

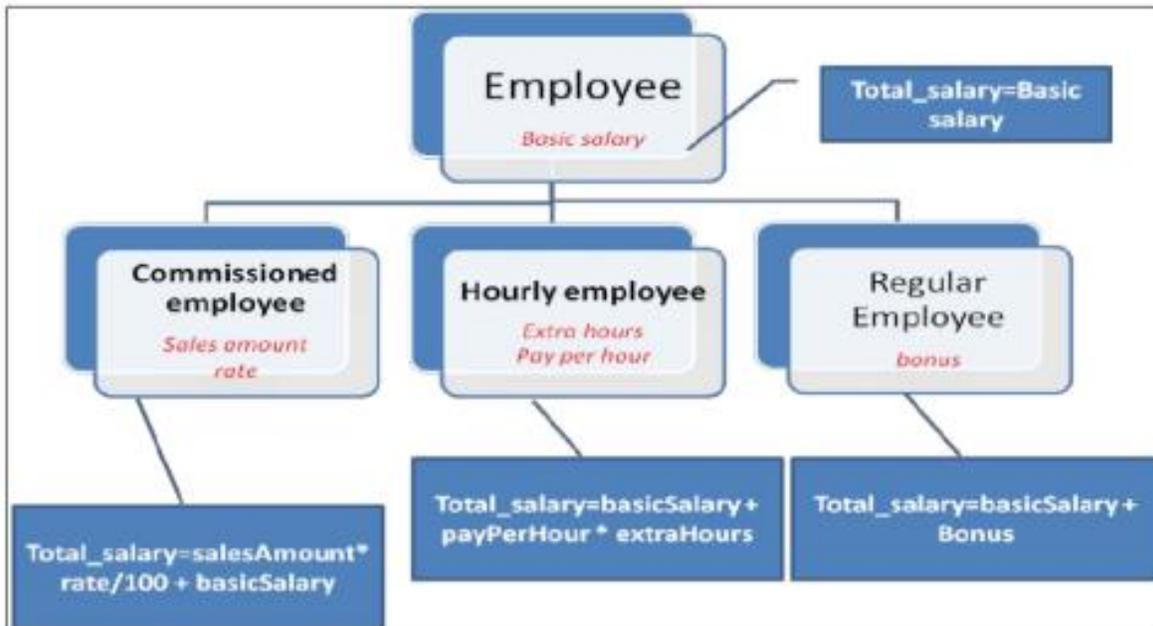
Example: What should be the output of this program?

```
class base{
public:
    virtual void who()    { cout<<"Base\n"; }
};
class first_d: public base{
public:
    void who()            { cout<<"First Derivation\n"; }
};
class second_d: public base{
public:
    void who()            { cout<<"Second Derivation\n"; }
};
int main(){
    base base_obj;
    base *p;
    first_d first_obj;
    second_d second_obj;
    p= & base_obj;
    p->who();
    p= & first_obj;
    p->who();
    p=& second_obj;
    p->who()
}
```

Exercise 1:

We want to design a system for a company to calculate salaries of different types of employees.

Consider the following diagram:



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:	Total Salary=sales amount*rate/100+basic salary
Hourly Employee:	Total salary=basic salary + pay per hour*extra hours
Regular Employee:	Total salary= basic salary + 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**

Lab Manual of Programming Fundamentals

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);

    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();
        cout<<endl;
    }

    return 0;
}
```

Post Lab:

Define a class **Shape** having an attribute *Area* and a pure virtual function *Calculate_Area*.

Also include following in this class.

- A constructor that initializes *Area* to zero.
- A method *Display()* that display value of member variable.

Now derive two classes from **Shape**; **Circle** having attribute *radius* and **Rectangle** having attributes *Length* and *Breadth*. Include following in each class.

- A constructor that takes values of member variables as argument.
- A method *Display()* that overrides *Display()* method of **Shape** class.
- A method *Calculate_Area()* that calculates the area as follows:

Area of Circle= $PI * Radius^2$

Area of Rectangle= $Length * Breadth$

Use following driver program to test above classes.

```
int main()
{
    Shape *p;
    Circle C1(5);
    Rectangle R1(4, 6);
    p=&C1;

    p->Calculate_Area();
    p->Display();

    p=&R1;
    p->Calculate_Area();
    p->Display();
    return 0;
}
```