

UNIT 6

VIRTUAL FUNCTION, POLYMORPHISM AND MISCELLANEOUS C++ FEATURES

LH – 5HRS

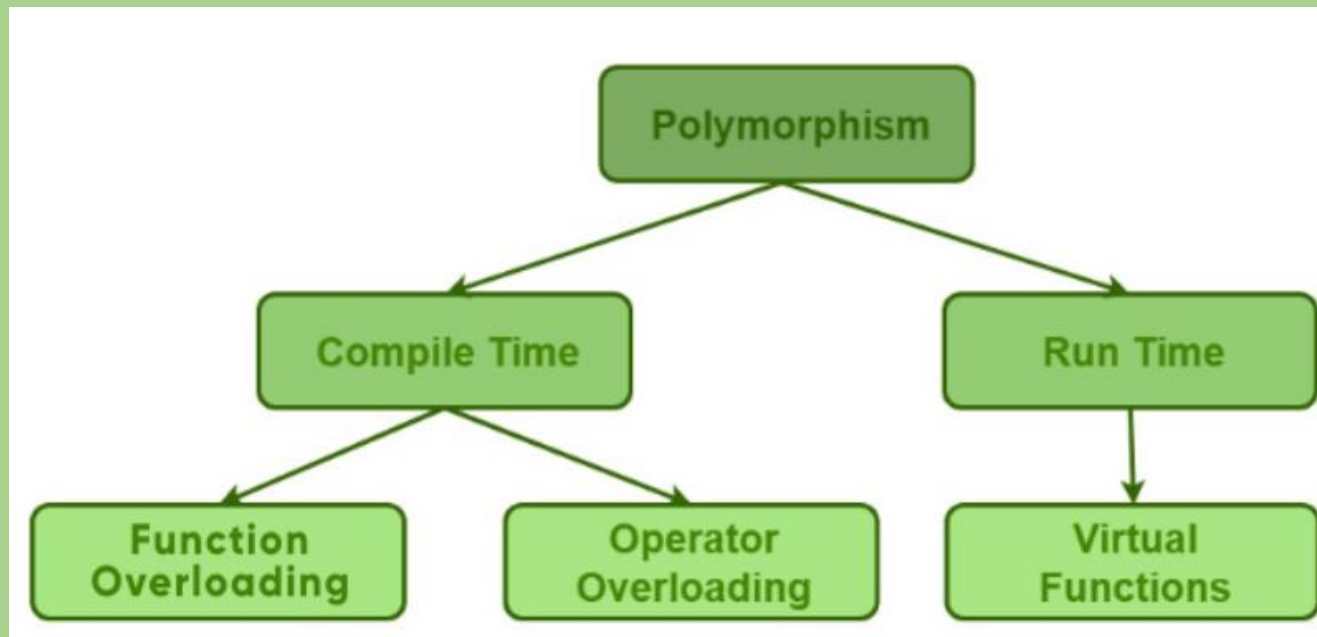
PRESENTED BY:
ER. SHARAT MAHARJAN
OOP IN C++

CONTENTS (LH – 5HRS)

- 6.1 Concept of Virtual functions (difference between normal member function accessed with pointers and virtual member function accessed with pointers),
- 6.2 Late Binding, Abstract class and pure virtual functions,
- 6.3 Virtual Destructors, Virtual base class,
- 6.4 Friend function, Friend class,
- 6.5 Static function
- 6.6 Assignment and copy initialization, Copy constructor
- 6.7 This pointer
- 6.8 Concrete classes (vs abstract class)
- 6.9 Polymorphism and its roles.

6.9 Polymorphism and its roles

- Polymorphism means state of **having many forms**. We have already seen that polymorphism is implemented using the concept of overloaded functions and operators.
- In this case, polymorphism is called **early binding** or **static binding** or **static linking**. This is also called **compile time polymorphism**.
- There is another kind of polymorphism called **run time polymorphism**. In this type, the selection of appropriate function is done dynamically at run time. So this is also called **late binding** or **dynamic binding**. C++ supports a mechanism known as **virtual functions** to achieve run time polymorphism. Run time polymorphism also requires the **use of pointers to objects of base class and method overriding**.



Role of Polymorphism:

1. Same interface could be used for creating methods with different implementations.
2. It helps programmers reuse the code and classes once written, tested and implemented.
3. Polymorphism helps in reducing the coupling between different functionalities.
4. Supports building extensible systems.

6.1 Concept of Virtual functions

- Virtual means existing in appearance but not in reality. When virtual functions are used, a program that appears to be calling a function of one class may in reality be calling a function of a different class.
- Furthermore, when we use virtual functions, different functions can be executed by the same function call statement.
- The information regarding which function to invoke is determined at run time. We can define virtual functions as below:

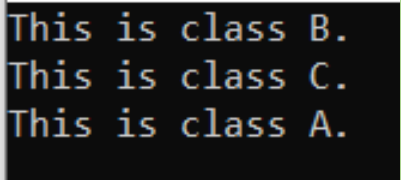
```
class A{  
    int x;  
    public:  
    virtual void display();  
};
```

Dynamic Polymorphism:

- We should use virtual functions and pointers to objects of base class to achieve run time polymorphism.
- For this, we use functions having same name, same number of parameters, and similar type of parameters in both base and derived classes i.e; function overriding.
- The function in the base class is declared as virtual using the keyword virtual.
- When a function in the base class is made virtual, C++ determines which function to use at run time based on the type of object pointed by the base class pointer, rather than the type of the pointer.

LAB 1: Virtual member function accessed with pointers

```
#include<iostream>
#include<conio.h>
using namespace std;
class A{
public:
    virtual void display(){
        cout<<"This is class A."<<endl;
    };
class B: public A{
public:
    void display(){
        cout<<"This is class B."<<endl;
    };
class C: public A{
public:
    void display(){
        cout<<"This is class C."<<endl;
    };
int main(){
    A *p,a;
    B b;
    C c;
    p=&b;
    p->display(); //points to Class B's object's function since Class A has display() virtual function.
    p=&c;
    p->display();
    p=&a;
    p->display();
    getch();
    return 0; }
```



```
This is class B.
This is class C.
This is class A.
```

LAB 2: Normal member function accessed with pointers

```
#include<iostream>
#include<conio.h>
using namespace std;
class A{
    public:
        void display(){
            cout<<"This is class A."<<endl;
        };
};
class B:public A{
    public:
        void display(){
            cout<<"This is class B."<<endl;
        };
};
class C:public A{
    public:
        void display(){
            cout<<"This is class C."<<endl;
        };
};
int main(){
    A *p,a;
    B b;
    C c;
    p=&b;
    p->display();
    p=&c;
    p->display();
    p=&a;
    p->display();
    getch();
    return 0; }
```

pardaina

OUTPUT:

```
This is class A.
This is class A.
This is class A.
```


6.2+6.8 Pure virtual functions, Abstract class and Concrete class

Pure virtual functions:

- A pure virtual function is one with the expression =0 added to the declaration.
- In such cases, the compiler requires each derived class to either define the function or redeclare it as a pure virtual function.
- Thus, a pure virtual function simply acts as a placeholder that is meant to be redefined by derived classes.

We can define pure virtual function as below:

```
class base{//abstract class- object can't be created of abstract class.
```

```
private:
```

```
int x,y;
```

```
public:
```

```
virtual void setData()=0;
```

```
virtual void getData()=0;};
```

```
class derived: public base{
```

```
    //Data members
```

```
    //Member functions};
```

```
//doesn't mean assignment of zero to the function. It is only the  
//specification for telling the compiler that the function is  
// pure virtual function.
```

Abstract class: (base class)

- An abstract class is a class containing at least one pure virtual function. They can't be instantiated into an object directly. However, we can create pointers and references to an abstract class. Abstract classes are designed to be specifically used as base classes. Only a subclass of an abstract class can be instantiated directly if all inherited pure virtual methods have been implemented by that class.

```
class A{  
    public:  
    fun1()=0;  
    fun2()=0;};
```

- With above abstract class, we can't create its object of but we can create pointer to A class as below:

```
A a; //invalid
```

```
A *p; //valid
```

Concrete class: (derived class)

- A derived class that implements all the missing functionality is called a concrete class..

LAB 3: Pure virtual function, abstract base class and concrete class

```
#include<iostream>
#include<conio.h>
using namespace std;
class Shape{ //abstract class
protected:
    int width, height;
public:
    void setData(int a, int b){
        width=a;
        height=b;
    }
    virtual int area()=0;      //pure virtual function
};
class Rectangle: public Shape{      //concrete class
public:
    int area(){
        return width*height;
    };
};
class Triangle: public Shape{      //concrete class
public:
    int area(){
        return (width*height)/2;
    };
};

int main(){
    Shape *s;
    Rectangle r;
    Triangle t;
    s=&r;
    s->setData(2,4);
    cout<<"Area of rectangle="<<s->area()<<endl;
    s=&t;
    s->setData(2,4);
    cout<<"Area of triangle="<<s->area()<<endl;
    getch();
    return 0; }
```

OUTPUT:

```
Area of rectangle=8
Area of triangle=4
```

6.3 Virtual Destructors, Virtual base class

- A virtual destructor ensures that when objects of derived subclasses go out of scope or are deleted, the order of destruction of each class in a hierarchy is carried out correctly.
- When we use the concept of virtual destructors, it forces the compiler to call the destructors of derived class while using base pointers with delete operator.
- Example given below explains the behavior of virtual vs non-virtual destructor.

//Virtual destructors and virtual base class

#include<iostream>

pardaina

#include<conio.h>

using namespace std;

class Base1{

public:

~Base1(){cout<<"Inside 1st base."<<endl;}

};

class Derived1: public Base1{

public:

~Derived1(){cout<<"Inside 1st derived."<<endl;}

};

class Base2{

public:

virtual ~Base2(){cout<<"Inside 2nd base."<<endl;}

};

class Derived2: public Base2{

public:

~Derived2(){cout<<"Inside 2nd derived."<<endl;}

};

int main(){

Base1 *b1=new Derived1;

Base2 *b2=new Derived2;

delete b1;

delete b2;

getch();

return 0;

}

OUTPUT:

```
Inside 1st base.  
Inside 2nd derived.  
Inside 2nd base.
```

6.4 Friend function, Friend class

Friend Class:

- A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class.

Friend Function:

A friend function can be given a special grant to access private and protected members.

- Following are some important points about friend functions and classes:
 - 1) Friends should be used only for limited purpose. Too many functions or external classes declared as friends of a class with protected or private data lessens the value of encapsulation of separate classes in object-oriented programming.
 - 2) Friendship is not mutual. If class A is a friend of B, then B doesn't become a friend of A automatically.
 - 3) Friendship is not inherited
 - 4) The concept of friends is not there in Java.

LAB 4: Friend Class

```
#include <iostream>
#include <conio.h>
using namespace std;
class A {
private:
    int x;
public:
    A() { a = 0; }
    friend class B; // Friend Class
};
class B {
private:
    int b;
public:
    void show(A a)
    {
        // Since B is friend of A, it can access private members of A
        cout << "Private member of class A accessed from friend class B: " << a.x;
    }
};
int main()
{
    A a;
    B b;
    b.show(a);
    getch();
    return 0;
}
```

OUTPUT:

```
Private member of class A accessed from friend class B: 0
-----
Process exited after 2.103 seconds with return value 0
Press any key to continue . . .
```

LAB 5: Friend Function

```
#include <iostream>
#include <conio.h>
using namespace std;
class sample{
    private:
        int a, b;
    public:
        void setData(){
            a=20;
            b=40;
        }
        friend float mean(sample s); //Friend function
};

float mean(sample s){
    return float(s.a+s.b)/2;
}

int main(){
    sample s;
    s.setData();
    cout<<"Mean of two numbers="<<mean(s);
    getch();
    return 0;
}
```


6.5 Static function

- One can create static member functions that work for the class as a whole rather than for a particular object of a class. One can call a static member function by using class itself along with scope resolution operator as:

```
class X{public: static void f(){}};
```

```
int main(){X::f();} //calling static member function
```

- A static member function can't access ordinary data members, only static data members. It can call only other static member functions.

LAB 6: Static function

```
#include <iostream>
#include <conio.h>
using namespace std;
class student{
    private:
        int roll;
        char name[20];
        static int count; //static data member
    public:
        void setData(){
            cout<<"Enter roll and name:"<<endl;
            cin>>roll>>name;
            count++;
        }
        static void getData(){ //static function
            cout<<"count="<<count<<endl;
        }
};
int student::count=0;
int main(){
    student s1,s2,s3;
    s1.setData();
    s2.setData();
    s3.setData();
    student::getData();
    getch();
    return 0;
}
```

OUTPUT:

```
Enter roll and name:
1 Ram
Enter roll and name:
2 Hari
Enter roll and name:
3 Sita
count=3
```

6.6 Assignment and copy initialization

- Copy constructor is called when a new object is created from an existing object, as a copy of the existing object. And assignment operator is called when an already initialized object is assigned a new value from another existing object. **Example:**

```
#include<iostream>
#include<stdio.h>
using namespace std;
class Test{
    public:
    Test() {}
    Test(Test &t){
        cout<<"Copy constructor called "<<endl;
    }
    void operator = (Test &t){
        cout<<"Assignment operator called "<<endl;
    };
};

int main(){
    Test t1, t2;
    t2 = t1;
    Test t3 = t1;
    getch();
    return 0;}
```

OUTPUT:

```
Assignment operator called
Copy constructor called
```

6.7 This pointer

- C++ uses a unique keyword called “this” to represent an object that invokes a member function.
- ‘this’ is a pointer that points to the object on which this function was called.

We can use this pointer for following two purposes:

- To resolve the name conflict between member variables and local variables in a method.
- To return the invoking object.

```
//This pointer
#include<iostream>
#include<stdio.h>
#include<string.h>
using namespace std;
class person{
    private:
        string name;
        int age;
    public:
        void setData(string name, int age){
            this->name=name;
            this->age=age;}
        void getData(){
            cout<<"Name:"<<name<<endl<<"Age:"<<age<<endl;}
        person isElder(person p){
            if(age>p.age)
                return *this;
            else
                return p;}};

int main(){
    person p,p1,p2;
    p1.setData("Ram",20);
    p2.setData("Sita",21);
    p=p1.isElder(p2);
    cout<<"Elder one is:"<<endl;
    p.getData();
    getchar();
    return 0;}

```

OUTPUT:

```
Elder one is:
Name:Sita
Age:21

```

THANK YOU FOR YOUR ATTENTION