

Unit6

Virtual Function, Polymorphism, and miscellaneous C++ Features

Polymorphism

- The word polymorphism means having many forms.
- Real life example of polymorphism:
 - a person at a same time can have different characteristic. Like a man at a same time is a father, a husband, a employee. So a same person posses have different behavior in different situations. This is called polymorphism.
- In 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.

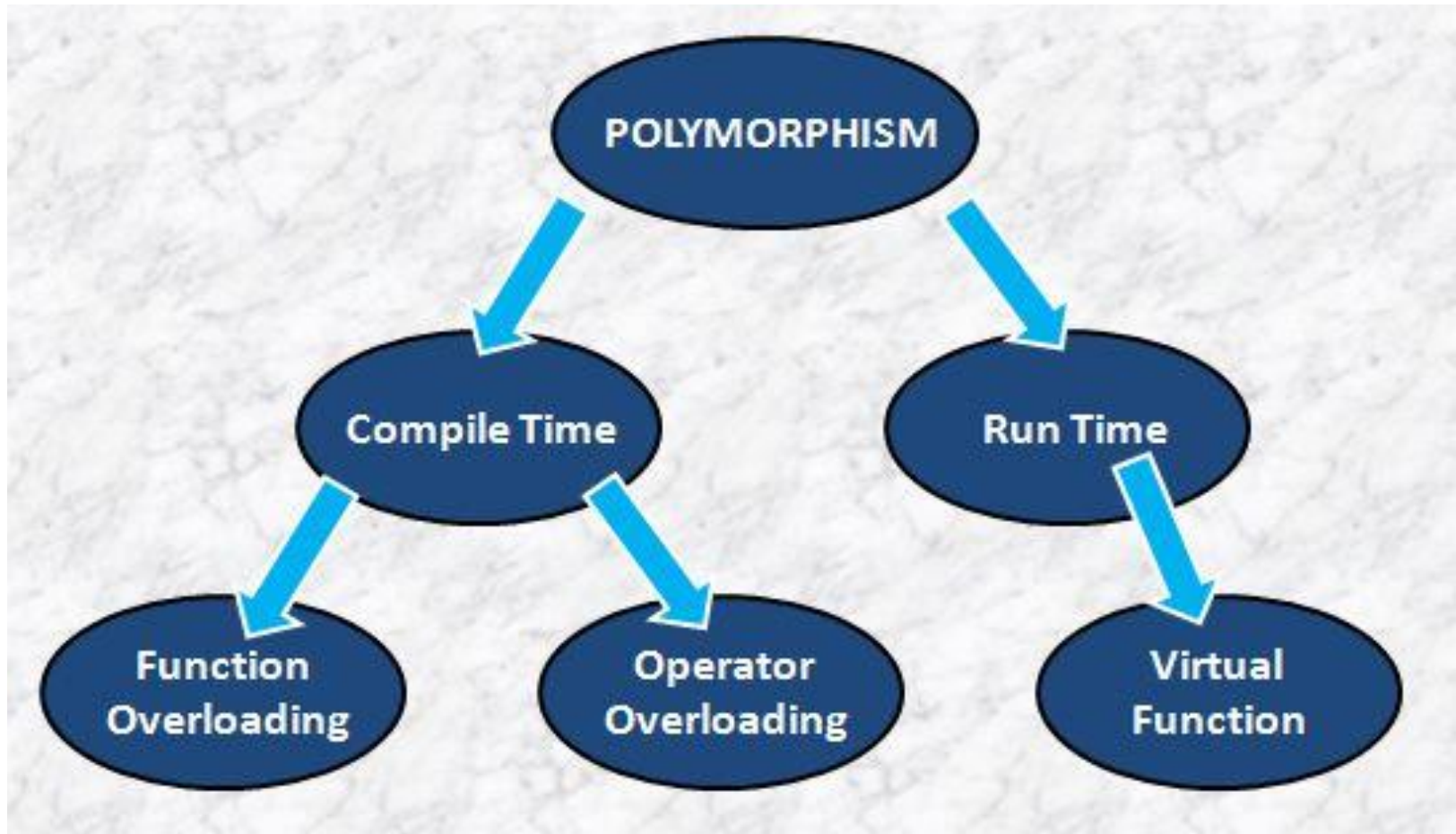
Contd..

- **Roles of polymorphism:**

- Same interface could be used for creating methods with different implementations.
- It helps programmers reuse the code once written, tested and implemented.
- Polymorphism helps in reducing the coupling between different functionalities.
- Support binding extensible systems.

Contd..

- In C++ polymorphism is mainly divided into two types:
 - Compile time Polymorphism/early binding/static binding
 - Runtime Polymorphism/late binding/dynamic binding



Contd..

- **Compile time Polymorphism/early binding/static binding:**
 - This is called compile time polymorphism because the compiler knows the information needed to call the function at the compile time and, therefore; compiler is able to select the appropriate function for a particular call at the compile time itself.
 - This type of polymorphism is achieved by function overloading, operator overloading.

Contd..

- Runtime Polymorphism/late binding/dynamic binding:
 - In this type, the selection of appropriate function is done dynamically at run time, so, this is also called late binding or dynamic binding.
 - This type of polymorphism is achieved by Function Overriding.
 - **Function overriding** occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

Contd..

- **Run time polymorphism?**

- can be achieved by defining virtual functions in base class using virtual keyword.
- Then, override the base class virtual function in derived class.
- Requires the use of pointer to objects of base class
- Such mechanism postpones the binding of function call to the member function declared as virtual for runtime.

Contd..

Compile time Polymorphism	Run time Polymorphism
In Compile time Polymorphism, call is resolved by the compiler .	In Run time Polymorphism, call is not resolved by the compiler.
It is also known as Static binding , Early binding and overloading as well.	It is also known as Dynamic binding , Late binding and overriding as well.
Overloading is compile time polymorphism where more than one methods share the same name with different parameters or signature and different return type.	Overriding is run time polymorphism having same method with same parameters or signature, but associated in a class & its subclass.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers .
It provides fast execution because known early at compile time.	It provides slow execution as compare to early binding because it is known at runtime.
Compile time polymorphism is less flexible as all things execute at compile time.	Run time polymorphism is more flexible as all things execute at run time.

Pointer revisited

- A **pointer** is a variable whose value is the address of another variable.
- **Pointer variables declaration:**
 - **Syntax:**
 - type *var-name;
 - **type** is the data type pointed to by the pointer. This type is not the type of the pointer itself, but the type of the data the pointer points to.
 - **Var_name** = name of pointer variable (should be valid identifier)
 - **asterisk (*)** used when declaring a pointer only means that it is a pointer

Contd..

- **For Example:**
 - `int *ip; // pointer to an integer`
 - `double *dp; // pointer to a double`
 - `float *fp; // pointer to a float`
 - `char *ch // pointer to character`

Contd..

- **Pointer initialization:**

- Pointer initialization to the address of a variable:

- For example:

```
int myvar;
```

```
int * myptr = &myvar;
```

- Pointer initialization to the value of another pointer:

- For example:

```
int myvar;
```

```
int *f = &myvar;
```

```
int *bar = f;
```

Contd..

- **Pointer to object:**

- The format for defining a pointer to an object is similar to defining pointer to standard data types.

- **Syntax:**

- `Class_name *po;`
 - `Class_name o;`
 - `Po=&o ;` **OR**
 - `po=new class_name;` (to release memory use `delete po;`)

- Syntax to access the member of objects:

- `Po->member;` **OR** `(*po).member;`

What is Virtual Functions?

- Virtual Function is a function in base class, which is overridden in the derived class, and which tells the compiler to perform **Late Binding** on this function.
- In simple words, a virtual function is a member function in the base class that we expect to redefine in derived class.

Need of virtual function

- Normal Member Functions Accessed with Pointers

```
//why concept of virtual function is necessary
#include<iostream>
using namespace std;
class animal
{
    public:
        void display()
        {
            cout<<"\n From base class animal";
        }
};
class cow:public animal
{
    void display()
    {
        cout<<"\n From derived class cow";
    }
};
class dog:public animal
{
    void display()
    {
        cout<<"\n From derived class dog";
    }
};
```

Contd..

```
int main()
{
    animal *pa; //pointer to base class
    animal a;
    cow c;
    dog d;

    pa=&a;
    pa->display();

    pa=&c;
    pa->display();

    pa=&d;
    pa->display();

    return 0;
}
```

Contd..

- When the above program is executed, it will produce the output as:

```
From base class animal  
From base class animal  
From base class animal
```

- What is happening in this case?
 - In this case during the compilation, it always selects the base class function(`display()`) as it is a pointer to the base class object.
 - The compiler is unknown about the address, which is known at runtime and chooses the member function that matches the type of pointer.
 - So when the program is executed the base class function is called.

Contd..

- so require a provision for selecting particular function as according to the address of the object the pointer holds.
 - To select a particular function as according to which objects it points the pointer should wait till the program executes.
 - i.e., Postponement of the decision of selecting a particular function until run-time.
 - Can be achieved by virtual function.

Virtual functions

- Virtual means existing in appearance/effect but not in reality.
- a virtual function defines a target function to be executed, but the target might not be known at compile time.
- A member function can be made as virtual function by preceding the member function with the keyword *virtual*.

syntax:

```
class test
{
    .....
    public:
        virtual return_type function_name(args...)
        {
            //function body
        }
};
```

Contd..

- Meaning of the virtual function becomes obvious if the virtual function is overridden and the base class pointer points(holds the addresses of) the derived class object.
- When such a declaration is made, it allows deciding function to be used at runtime, based on the type of object, pointed to by the base pointer, rather than the type of the pointer.

Contd..

- Virtual Member Functions Accessed with Pointers

//why concept of virtual function is necessary

```
#include<iostream>
using namespace std;
class animal
{
    public:
        virtual void display()
        {
            cout<<"\n From base class animal";
        }
};
class cow:public animal
{
    void display()
    {
        cout<<"\n From derived class cow";
    }
};
class dog:public animal
{
    void display()
    {
        cout<<"\n From derived class dog";
    }
};
```

Contd..

```
int main()
{
    animal *pa; //pointer to base class
    animal a;
    cow c;
    dog d;

    pa=&a;
    pa->display();

    pa=&c;
    pa->display();

    pa=&d;
    pa->display();

    return 0;
}
```

```
From base class animal
From derived class cow
From derived class dog
```

Abstract classes and pure virtual functions

- When object of base class are never instantiated, such a class is called abstract base class or simply abstract class.
- An abstract class can be used only as an interface for class hierarchy and as a base for other classes.
- Abstract classes are created when creating class hierarchy with virtual functions because base pointers are used but the base class objects are rarely created and the base class virtual function are not called.
- To make abstract class define at least one pure virtual function in it.

Contd..

- **Pure virtual function:**

- A pure virtual function is one with the expression `=0` added to the declaration. i.e., A virtual function with null body is called a pure virtual function.
- The syntax of declaration of pure virtual function and making a class abstract is:

```
class abstract_class_name
{
    public:
        virtual return_type function_name()=0; //pure virtual function
};|
```

Contd..

- Here, class `abstract_class_name` become abstract since there is presence of pure virtual function..
- The expression `=0` does not have any other meaning except the function has a null body. The equal sign `=0` does not assign zero to function. It is only a method to tell the compiler that the function is pure virtual function and class `abstract_class_name` is abstract class.
- **Concrete class**: class without pure virtual function is called concrete class. We can create object of concrete class.


```

//abstract class and pure virtual functions
#include<iostream>
using namespace std;
class polygon    //abstract class
{
    protected:
        int width,height;
    public:
        void setvalues(int a, int b)
        {
            width=a;
            height=b;
        }
        virtual int area()=0;
};
class rectangle:public polygon    //concrete class
{
    public:
        int area()
        {
            return(width*height);
        }
};
class triangle:public polygon    //concrete class
{
    public:
        int area()
        {
            return (width*height/2);
        }
};

```

Contd..

```
int main()
{
    rectangle r;
    triangle t;
    polygon *p=&r;
    p->setvalues(4,5);
    cout<<"Area of recrangle="<<p->area()<<endl;

    p=&t;
    p->setvalues(6,5);
    cout<<"Area of Triangle="<<p->area()<<endl;
    return 0;
}
```

```
Area of recrangle=20
Area of Triangle=15
```

Virtual destructor

- Deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behavior.
- To correct this situation, the base class should be defined with a virtual destructor.

```
syntax:
class base_class_name
{
    .....
public:
    .....
    virtual ~base_class_name()
    {
        //body
    }
};
```

- When we call the virtual destructor, it forces the compiler to call the destructors of derived class while using base pointers with **delete** operator.

Contd..

```
// A program with virtual destructor
#include<iostream>
using namespace std;
class base
{
public:
    base()
    {
        cout<<"Constructing base \n";
    }
    virtual ~base()
    {
        cout<<"Destructing base \n";
    }
};

class derived: public base
{
public:
    derived()
    {
        cout<<"Constructing derived \n";
    }
    ~derived()
    {
        cout<<"Destructing derived \n";
    }
};
```

Contd..

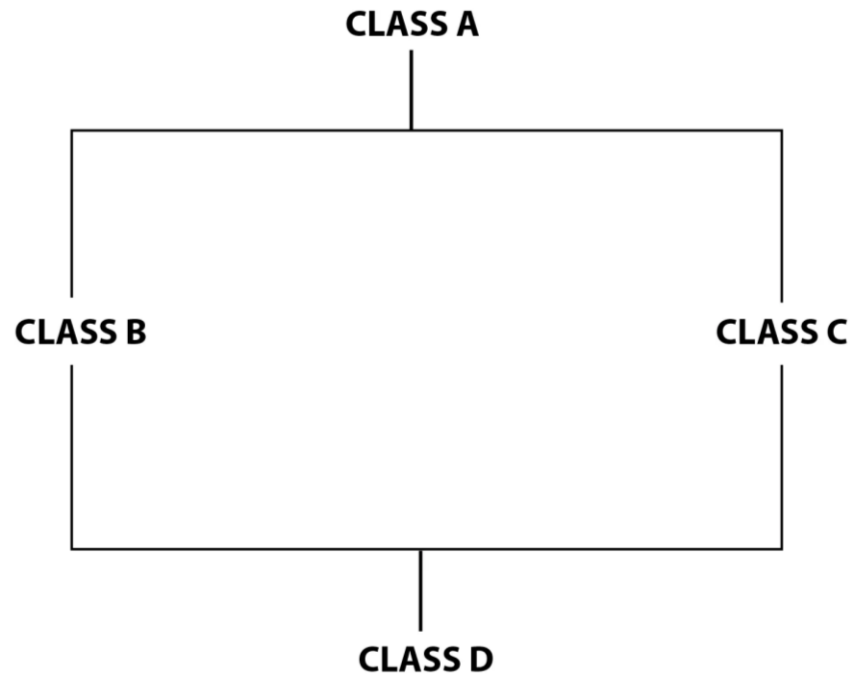
```
int main(void)
{
    derived *d = new derived();
    base *b = d;
    delete b;
    return 0;
}
```

- **Output:**

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

Virtual Class

- Virtual base classes in C++ are used to prevent multiple instances of a given class from appearing in an inheritance hierarchy when multiple inheritances are used.



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

class A {
public:
    A() {
        cout << "Constructor A\n";
    }
    void display() {
        cout << "Hello form Class A \n";
    }
};

class B: public A {
};

class C: public A {
};

class D: public B, public C {
};
```

```
int main() {
    D object;
    object.display();
}
```

- Error is seen due to ambiguous paths of derivation.

ERROR!

```
/tmp/5LQZXN5yM7.cpp: In function 'int main()':
```

```
/tmp/5LQZXN5yM7.cpp:25:10: error: request for member 'display' is  
ambiguous
```

```
25 |     object.display();  
    |           ^~~~~~
```

```
/tmp/5LQZXN5yM7.cpp:9:10: note: candidates are: 'void A::display()'
```

```
9 |     void display() {  
  |           ^~~~~~
```

```
/tmp/5LQZXN5yM7.cpp:9:10: note:                               'void A::display()'
```


- After the `object.display()` line is removed, the code produces following output.

```
/tmp/tgyXmAX1n1.o  
Constructor A  
Constructor A
```

- It means that the base class is called multiple times hence the condition is known as ambiguous.
- This can be solved using the virtual class.

```

#include <iostream>
using namespace std;

class A {
public:
    A() {
        cout << "Constructor A\n";
    }
    void display() {
        cout << "Hello form Class A \n";
    }
};

class B: public virtual A {
};

class C: public virtual A {
};

class D: public B, public C {
};

```

- The base class can be made virtual by using the ‘virtual’ keyword before the class name during derivation.
- ‘public virtual’ and ‘virtual public’ are same.

```

int main() {
    D object;
    object.display();
}

```

- Output:

```

Constructor A
Hello form Class A

```

- Using the keyword `virtual` ensures that only a single copy of members of class A is inherited to class D.

Friend Functions

- The encapsulation and data hiding do not allow non-member functions to access the object's private data.
- If need to access the private member of the object then must make a public member function.
- But inconvenience!
- How a outer function operate on private members of a class and how a function operate on objects of two different classes?
- This can be achieved by using the concept of **friend function**.

Contd..

- To make an outside function friendly to a class, we simply declare the function as a friend of the class by using the **friend keyword** as shown below:

```
class class_name
{
    private:
        .....
    public:
        //.....
        friend return_type function_name( args); //declaration
};
```

- a friend function is explicitly declared in the body of the class of which it is friend.
- Friend function can be placed in any section(private, public or protected).
- The function mentioned as a friend of a class can be **a global function** or **member function** of another class.

Contd..

- Some special characteristics of friend functions are:
 - Since it is not in the scope of the class to which it has been declared as a friend, it can not be called using the object of the class.
 - It can be invoked like a normal function without the help of any object.
 - Unlike member functions, it can not access the member name directly and has to use an object and dot membership operator with each member name.
 - Usually, it has the objects as arguments.

• **Example1:**

Contd..

```
//Friend function
#include<iostream>
using namespace std;
class sample
{
    private:
        int a ,b;
    public:
        void setValue()
        {
            a= 25;
            b= 40;
        }
        friend float mean(sample s);
};

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

int main()
{
    sample x;
    x.setValue();
    cout<<"Mean value="<<mean(x);
    return 0;
}
```

Mean value=32.5

Contd..

- If we want to operate on objects of two different classes, we must make a function that is friend in both the classes.

```
class second;    //declaration like function prototype
class first
{
    private:
        .....
    public:
        //.....
        friend return_type function_name( first f, second s); //friend function declaration
};

class second
{
    private:
        .....
    public:
        //.....
        friend return_type function_name( first f, second s); //friend function declaration
};

friend function definition
return_type function_name(first f, second s)
{
    //body of friend function
}
```


Contd..

- Example2:

```
//Friend functions
#include<iostream>
using namespace std;
class beta;
class alpha
{
private:
    int data;
public:
    void setdata(int d)
    {
        data=d;
    }
    friend int sum(alpha, beta);
};
class beta
{
private:
    int data;
public:
    void setdata(int d)
    {
        data= d;
    }
    friend int sum(alpha, beta);
};
```

```
int sum(alpha a, beta b)
{
    return a.data+b.data;
}
int main()
{
    alpha a;
    a.setdata(7);
    beta b;
    b.setdata(3);
    cout<<"sum="<<sum(a,b);
    return 0;
}
```

SUM=10

Contd..

- The member of another class is declared as friend as follows:

```
class A; //declaration of class A
class B
{
    //.....
    friend return_type A::function_name();
}
```

- Any function of another class can be declared as friend of one.
- If we need to make all function of another class as friend of one then the class as a whole is declared as friend.

Friend classes

- A class can also be declared to be a friend of some other class. When we create a friend class then all the member functions of the friend class also become the friend of the other class.

```
syntax:  
    class A;  
    class B  
    {  
        .....  
        friend class A;  
    };
```

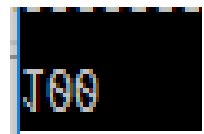
- **Example:**

Contd..

```
//Friend class
#include<iostream>
using namespace std;
class beta;
class alpha
{
    private:
        int x;
    public:
        void setdata(int d)
        {
            x=d;
        }
        friend class beta;
};
class beta
{
    public:
        void fun(alpha a)
        {
            cout<<a.x;
        }
};
```

```
int main()
{
    alpha a;
    a.setdata(100);
    beta b;
    b.fun(a);
    return 0;
}
```

Output:

A screenshot of a terminal window with a black background and white text. The text '100' is displayed on a single line, representing the output of the program.

This pointer

- The non-static member function invocation can be done in two way:
 - Without the object of the class
 - With the object of the class
- When non-static member function is invoked with the object of class then the **this pointer** comes into existence with the value of the address of the object through which it is called and is a member.
 - This pointer is a pointer which points to the object itself i.e., object involved in the invocation of non-static member function.
- **this pointer** is passed as implicit argument to the non-static member function apart from the explicit argument passed to it.
- Thus any non-static member function can find out the address to the object of which it is a member of.

Contd..

for example:

```
class Test
{
    private:
        int data;
    public:
        void setdata(int n)
        {
            data= n;    //this->data=n
        }
};
```

```
int main()
{
    Test t1;
    t1.setdata(5);
}
```

// the this pointer in setdata() function

// in this case will have the address of the object t1

Contd..

- We can use this pointer for following purposes:
 - To resolve the name conflict between member variables and local variables in a method.
 - To return the invoking object.

Contd..

```
//This pointer
#include<iostream>
#include<string.h>
using namespace std;
class person
{
    private:
        char name[20];
        int age;
    public:
        void setdata( char name[], int age)
        {
            strcpy(this->name,name);    //name conflict resolution
            this->age=age;
        }
        void display()
        {
            cout<<"Name:"<<this->name<<endl;
            cout<<"Age:"<<this->age<<endl;
        }
        person isElder(person p)
        {
            if(age>p.age)
                return *this;    //retruning invoking object
            else
                return p;
        }
};
```


Contd..

```
int main()
{
    person p, p1, p2;
    p1.setdata("Ram", 10);
    p2.setdata("Hari", 20);
    p = p1.isElder(p2);
    cout<<"Elder one is:"<<endl;
    p.display();
    return 0;
}
```

- **Output:**

```
Elder one is:
Name: Hari
Age: 20
```

- **Example 2:**

Contd..

```
//Example of "this" pointer:
```

```
#include <iostream>
```

```
using namespace std;
```

```
class X
```

```
{
```

```
    private:
```

```
        int a;
```

```
    public:
```

```
        void Set_a(int a)
```

```
        {
```

```
            // The 'this' pointer is used to retrieve 'xobj.a'
            // hidden by the automatic variable 'a'
```

```
            this->a = a;
```

```
        }
```

```
        void Print_a()
```

```
        {
```

```
            cout << "a = " << a << endl;
```

```
        }
```

```
};
```

```
int main()
```

```
{
```

```
    X xobj;
```

```
    int a = 5;
```

```
    xobj.Set_a(a);
```

```
    xobj.Print_a();
```

```
}
```

OUTPUT:

```
a = 5
```

Homework

- Explain various methods of polymorphism in C++.How do virtual function accomplish run time polymorphism? Write a simple program to demonstrate virtual function.
- What is polymorphism? Distinguish runtime polymorphism with compile time polymorphism.
- What do you mean by virtual function? How can it be declared? Why is virtual function necessary?
- What do you mean by abstract class? Explain the need for abstract class while building class hierarchy?
- What is pure virtual function? How do they differ from normal virtual function? Explain.
- What do you mean by virtual destructor? Explain with suitable example why virtual destructor is necessary.
- Illustrate the overloading of copy constructor using the assignment operator.
- Explain the use of this pointer with suitable example.
- Does friend function violate data hiding? Explain with an example. What are the some advantages and disadvantages of using friend functions?

Thank You !