### Polymorphism

Polymorphism means having multiple forms of one thing. In inheritance, polymorphism is done, by method overriding, when both super and sub class have member function with same declaration but different definition.

#### Function Overriding

If we inherit a class into the derived class and provide a definition for one of the base class's function again inside the derived class, then that function is said to be **overridden**, and this mechanism is called **Function Overriding**

#### Requirements for Overriding

1. Inheritance should be there. Function overriding cannot be done within a class. For this we require a derived class and a base class.
2. Function that is redefined must have exactly the same declaration in both base and derived class, that means same name, same return type and same parameter list.

#### Example of Function Overriding

class Base

{

public:

void show()

{

cout << "Base class";

}

};

class Derived:public Base

{

public:

void show()

{

cout << "Derived Class";

}

}

In this example, function **show()** is overridden in the derived class. Now let us study how these overridden functions are called in **main()** function.

#### Function Call Binding with class Objects

Connecting the function call to the function body is called **Binding**. When it is done before the program is run, its called **Early** Binding or **Static** Binding or **Compile-time** Binding.

class Base

{

public:

void shaow()

{

cout << "Base class\t";

}

};

class Derived:public Base

{

public:

void show()

{

cout << "Derived Class";

}

}

int mian()

{

Base b; //Base class object

Derived d; //Derived class object

b.show(); //Early Binding Ocuurs

d.show();

}

Output : Base class    Derived class

In the above example, we are calling the overrided function using Base class and Derived class object. Base class object will call base version of the function and derived class's object will call the derived version of the function.

#### Function Call Binding using Base class Pointer

But when we use a Base class's pointer or reference to hold Derived class's object, then Function call Binding gives some unexpected results.

class Base

{

public:

void shaow()

{

cout << "Base class";

}

};

class Derived:public Base

{

public:

void show()

{

cout << "Derived Class";

}

}

int main()

{

Base\* b; //Base class pointer

Derived d; //Derived class object

b = &d;

b->show(); //Early Binding Ocuurs

}

Output : Base class

In the above example, although, the object is of Derived class, still Base class's method is called. This happens due to Early Binding.

Compiler on seeing **Base class's pointer**, set call to Base class's **show()** function, without knowing the actual object type.

### Virtual Functions

Virtual Function is a function in base class, which is overrided in the derived class, and which tells the compiler to perform **Late Binding** on this function.

Virtual Keyword is used to make a member function of the base class Virtual.

#### Late Binding

In Late Binding function call is resolved at runtime. Hence, now compiler determines the type of object at runtime, and then binds the function call. Late Binding is also called **Dynamic** Binding or **Runtime** Binding.

#### Problem without Virtual Keyword

class Base

{

public:

void show()

{

cout << "Base class";

}

};

class Derived:public Base

{

public:

void show()

{

cout << "Derived Class";

}

}

int main()

{

Base\* b; *//Base class pointer*

Derived d; *//Derived class object*

b = &d;

b->show(); *//Early Binding Ocuurs*

}

Output : Base class

When we use Base class's pointer to hold Derived class's object, base class pointer or reference will always call the base version of the function

#### Using Virtual Keyword

We can make base class's methods virtual by using **virtual** keyword while declaring them. Virtual keyword will lead to Late Binding of that method.

class Base

{

public:

**virtual** void show()

{

cout << "Base class";

}

};

class Derived:public Base

{

public:

void show()

{

cout << "Derived Class";

}

}

int main()

{

Base\* b; *//Base class pointer*

Derived d; *//Derived class object*

b = &d;

b->show(); *//Late Binding Ocuurs*

}

Output : Derived class

On using Virtual keyword with Base class's function, Late Binding takes place and the derived version of function will be called, because base class pointer pointes to Derived class object.

#### Using Virtual Keyword and Accessing Private Method of Derived class

We can call **private** function of derived class from the base class pointer with the help of virtual keyword. Compiler checks for access specifier only at compile time. So at run time when late binding occurs it does not check whether we are calling the private function or public function.

#include

using namespace std;

class A

{

public:

**virtual** void show()

{

cout << "Base class\n";

}

};

class B: public A

{

private:

**virtual** void show()

{

cout << "Derived class\n";

}

};

int main()

{

A \*a;

B b;

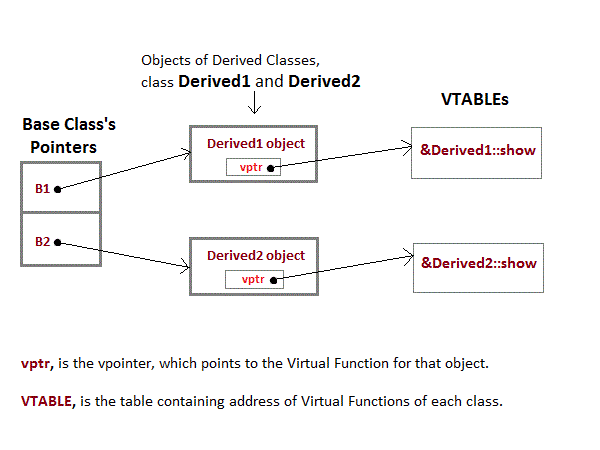
a = &b;

a **->** show();

}

Output : Derived class

#### Mechanism of Late Binding



To accomplich late binding, Compiler creates **VTABLEs**, for each class with virtual function. The address of virtual functions is inserted into these tables. Whenever an object of such class is created the compiler secretly inserts a pointer called **vpointer**, pointing to VTABLE for that object. Hence when function is called, compiler is able to resovle the call by binding the correct function using the vpointer.

#### Important Points to Remember

1. Only the Base class Method's declaration needs the **Virtual** Keyword, not the definition.
2. If a function is declared as **virtual** in the base class, it will be virtual in all its derived classes.
3. The address of the virtual Function is placed in the **VTABLE** and the copiler uses **VPTR**(vpointer) to point to the Virtual Function.