### **What is Polymorphism?**

In OOPs, polymorphism refers to the ability of objects to have names but different functionality. The most common use is with base class pointers or references calling derived class functions. This is called runtime polymorphism and is implemented in C++ using virtual functions.

### **Virtual Functions in C++**

A virtual function is a member function that is declared within a base class with the virtual keyword and is re-defined (Overridden) by a derived class. When a class contains a virtual function, it can be overridden in its derived class and tells the compiler to perform dynamic linkage (or late binding) on the function. Now, the virtual functions are implemented in C++ by using vTable and vPtr.

## **What is vTable?**

The vTable, or Virtual Table, is a table of function pointers that is created by the compiler to support dynamic polymorphism. Whenever a class contains a virtual function, the compiler creates a Vtable for that class. Each object of the class is then provided with a hidden pointer to this table, known as Vptr.

The Vtable has one entry for each virtual function accessible by the class. These entries are pointers to the **most derived function** that the current object should call.

## **What is vPtr (Virtual Pointer)?**

**The virtual pointer or \_vptr** is a hidden pointer that is added by the compiler as a member of the class to point to the VTable of that class. For every object of a class containing virtual functions, a vptr is added to point to the vTable of that class. It’s important to note that vptr is created only if a class has or inherits a virtual function.

## **Understanding vTable and vPtr Using an Example**

Here is a simple example with a base class **Base** and classes **Derived1** derived from Base and **Derived2** from class **Derived1**

// C++ program to show the working of vtable and vptr

#include <iostream>

using namespace std;

// base class

class Base {

public:

virtual void function1()

{

cout << "Base function1()" << endl;

}

virtual void function2()

{

cout << "Base function2()" << endl;

}

virtual void function3()

{

cout << "Base function3()" << endl;

}

};

// class derived from Base

class Derived1 : public Base {

public:

// overriding function1()

void function1()

{

cout << "Derived1 function1()" << endl;

}

// not overriding function2() and function3()

};

// class derived from Derived1

class Derived2 : public Derived1 {

public:

// again overriding function2()

void function2()

{

cout << "Derived2 function2()" << endl;

}

// not overriding function1() and function3()

};

// driver code

int main()

{

// defining base class pointers

Base\* ptr1 = new Base();

Base\* ptr2 = new Derived1();

Base\* ptr3 = new Derived2();

// calling all functions

ptr1->function1();

ptr1->function2();

ptr1->function3();

ptr2->function1();

ptr2->function2();

ptr2->function3();

ptr3->function1();

ptr3->function2();

ptr3->function3();

// deleting objects

delete ptr1;

delete ptr2;

delete ptr3;

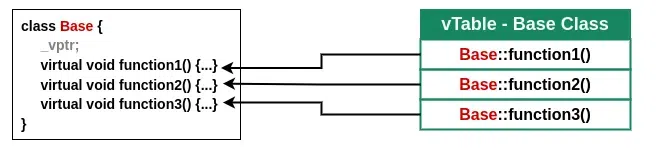
return 0;

}

### **Explanation**

**1. For Base Class**

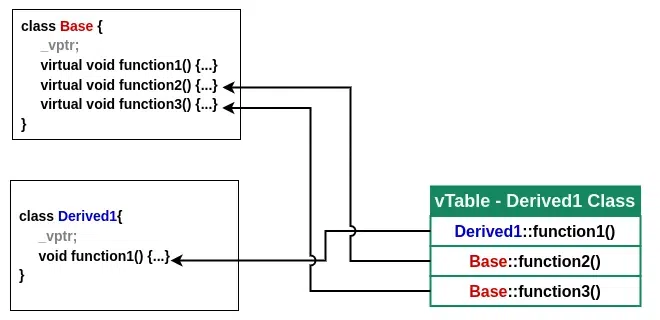
The **base class** have 3 virtual function **function1(), function2() and function3().** So the vTable of the base class would have 3 elements i.e. function pointer to base::function()



*Base Class vTable*

**2. For Derived1 Class**

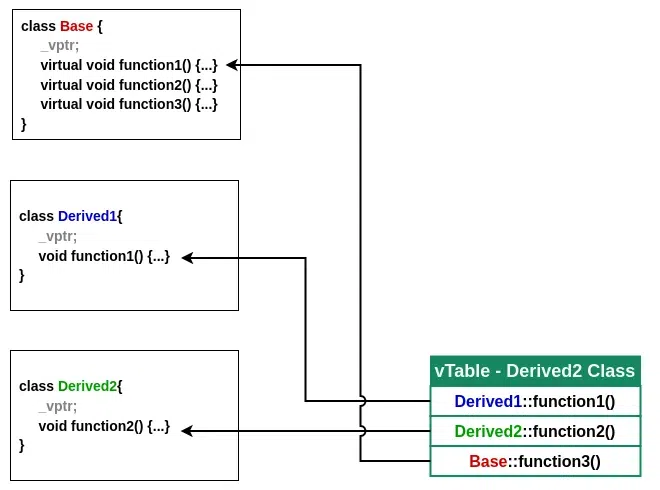
The D**erived1** class inherits 3 virtual functions from the **Base** class but overrides only **function1()** so all the other functions will be the same as the base class.



*Derived1 Class vTable*

**3. For Derived2 Class**

The **Derived2** class is inherited from **Derived1** class so it inherits the **function1() of Derived1** and **function2() and function3() of Base** class. In this class, we have overridden only function2(). So, the vTable for Derived2 class will look like this



*Derived2 Class vTable*