Virtual Functions

Contents

[1. Virtual Functions 2](#_Toc427013956)

[a. Virtual Table 2](#_Toc427013957)

[b. \_vptr 2](#_Toc427013958)

# Virtual Functions

## Virtual Table

* Virtual Table is a ***lookup table*** of function pointers.
* Virtual Table is used to dynamically bind the **virtual functions** to objects at runtime.
* Every class that *uses virtual functions* or *if a class is derived from a class that uses virtual functions*, is given it's own virtual table as a secret data member.
* This table is set up by the compiler at compile time.
* A virtual table contains one entry as a function pointer **for each virtual function** that can be called by objects of the class.
* Virtual table stores NULL pointer to **pure virtual functions** in ABC.
* Virtual Table is created even for classes that have virtual base classes. In this case, the vtable has pointer to the shared instance of the base class along with the pointers to the classe's virtual functions if any.

## \_vptr

* This **vtable pointer** or **\_vptr**, is a hidden pointer added by the Compiler to the base class.
* This pointer is **pointing** to the virtual table of that particular class.
* This **\_vptr** is **inherited** to all the derived classes.
* **Each object** of a *class with virtual functions* transparently stores this **\_vptr**.
* Call to a virtual function by an object is resolved by following this hidden \_vptr.

Example: Here we have 3 classes Base, DerivedOne and DerivedTwo. Where DerivedOne and DerivedTwo are derived from class Base.

#include <iostream>

using namespace std;

class Base

{

public:

virtual void function1() { cout << "Base :: function1()\n"; };

virtual void function2() { cout << "Base :: function2()\n"; };

virtual ~Base(){};

};

class DerivedOne : public Base

{

public:

~DerivedOne(){};

virtual void function1() { cout << "DerivedOne :: function1()\n"; };

};

class DerivedTwo : public Base

{

public:

~DerivedTwo(){};

virtual void function2() { cout << "DerivedTwo :: function2\n"; };

};

int main()

{

DerivedOne \*d = new DerivedOne;

Base \*b = d;

b->function1();

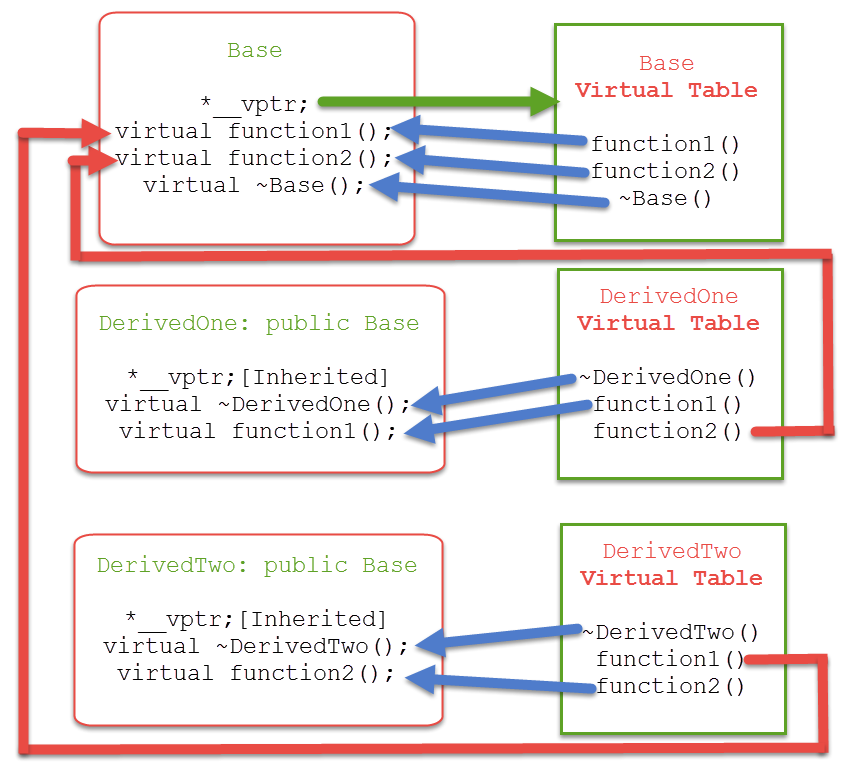
b->function2();

delete (b);

return (0);

}

A pictorial representation of Virtual Table and \_vptr for the above code:



Expalnation :

* Here in function main b pointer gets assigned to DerivedOne's **\_vptr** and now starts pointing to DerivedOne's vtable.
* Then calling to a function1(), makes it's **\_vptr** startightway calls DerivedOne's **vtable** function1() and so in turn calls DerivedOne's method i.e. function1() as DerivedOne has it's own function1() defined it's class.
* Where as pointer b calling to a function2(), makes it's **\_vptr** points to DerivedOne's **vatble** which in-turn pointing to Base class's **vtable** function2() as shown in the diagram (as DerivedOne class does not have it's own definition or function2()).
* So, now calling delete on pointer b follows the **\_vptr** - which is pointing to DerivedOne's **vtable** calls it's own class's destructor i.e. DerivedOne class's destructor and then calls the destrcutor of Base class - this as part of when dervied object gets deleted it turn deletes it's emebeded base object. Thats why we must always make Base class's destrcutor as virtual if it has any virtual functions in it.