Introduction

This article talks about how **virtual table** or **vtable**and **_vptr** works, with a pictorial representation.

Background

Virtual Table is a lookup table of function pointers used to dynamically bind the virtual functions to objects at runtime. It is not intended to be used directly by the program, and as such there is no standardized way to access it.

Virtual Table:

Every class that uses virtual functions (or 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. And 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.

Here is a simple example with a vtable representation.

Here we have 3 classes Base, D1 and D2. Where D1 and D2 are derived from class Base.

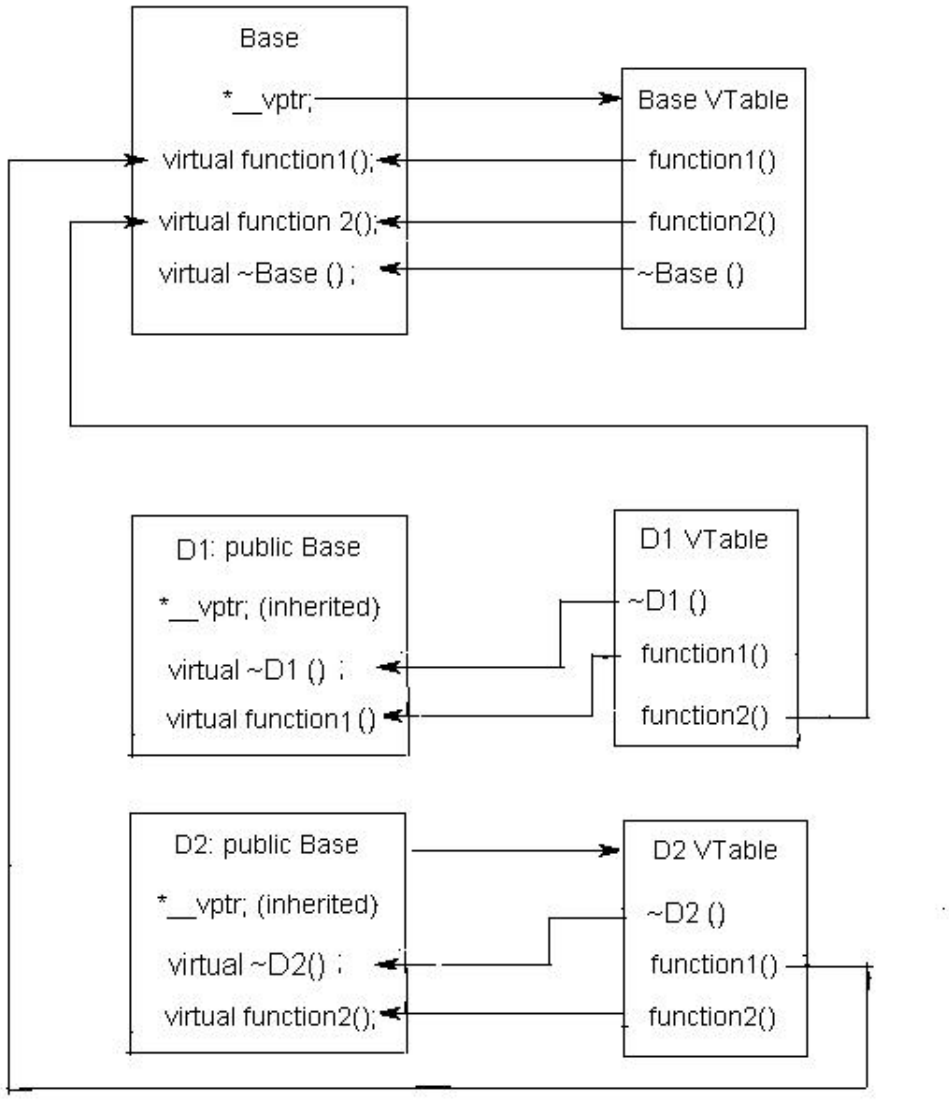
The Code

Code:

```
#include<iostream.h>

class Base
 {
 public:
    virtual void function1() {cout<<"Base :: function1()\n";};
    virtual void function2() {cout<<"Base :: function2()\n";};

    virtual ~Base(){};
```

```
};

class D1: public Base
{
public:
   ~D1(){};
   virtual void function1() { cout<<"D1 :: function1()\n";};
};

class D2: public Base
{
public:
   ~D2(){};
   virtual void function2() { cout<< "D2 :: function2\n";};
};

int main()
{
  D1 *d = new D1;;
  Base *b = d;

  b->function1();
  b->function2();
```

Here is a pictorial representation of Virtual Table and _vptr for the above code:



Expalnation :

Here in function main **b** pointer gets assigned to D1's _vptr and now starts pointing to D1's vtable. Then calling to a **function1()**, makes it's _vptr startightway calls D1's vtable **function1()** and so in turn calls D1's method i.e. **function1()** as D1 has it's own **function1()** defined it's class.

Where as pointer **b** calling to a **function2()**, makes it's _vptr points to D1's vatble which in-turn pointing to **Base** class's vtable **function2 ()** as shown in the diagram (as D1 class does not have it's own definition or **function2()**).

So, now calling delete on pointer **b** follows the _vptr - which is pointing to D1's vtable calls it's own class's destructor i.e. D1 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.**

Bootom Line:
Thats how the Run-time or Dynamic binding happens on calling virtual functions of different derived objects.