VIRTUAL FUNCTION IN CPP

Virtual Functions

A*virtual function* is a member function that is declared within a base class and redefined by a derived class. To create a virtual function, precede the function's declaration in the base class with the keyword **virtual**. When a class containing a virtual function is inherited, the derived class redefines the virtual function to fit its own needs. In essence, virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism.

The virtual function within the base class defines the *form* of the *interface* to that function. Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a *specific method*. When accessed "normally," virtual functions behave just like any other type of class member function. However, what makes virtual functions important and capable of supporting run-time polymorphism is how they behave when accessed via a pointer. As discussed in earlier, a base-class pointer can be used to point to an object of any class derived from that base. When a base pointer points to a derived object that contains a virtual function, C++ determines which version of that function to call based upon *the type of object pointed to* by the pointer. And this determination is made *at run time*. Thus, when different objects are pointed to, different versions of the virtual function are executed. The same effect applies to base-class references. To begin, examine this short example:

```
#include   <iostream>
using namespace std;
class base {
public:
virtual void vfunc() {
cout << "This is base's vfunc().\n";
}
};
class derived1 : public base {
public:
void vfunc() {
```

```
cout << "This is derived1's vfunc().\n";
}
};
class derived2 : public base {
public:
void vfunc() {
cout << "This is derived2's vfunc().\n";
}
};
int main()
{
base *p, b;
derived1 d1;
derived2 d2;
// point to base
p = &b;
p->vfunc(); // access base's vfunc()
// point to derived1
p = &d1;
p->vfunc(); // access derived1's vfunc()
// point to derived2
p = &d2;
p->vfunc(); // access derived2's vfunc()
return 0;
}
```

This program displays the following:

```
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().
```

As the program illustrates, inside **base**, the virtual function **vfunc( )** is declared.

Notice that the keyword **virtual** precedes the rest of the function declaration. When **vfunc( )** is redefined by **derived1** and **derived2**, the keyword **virtual** is not needed. (However, it is not an error to include it when redefining a virtual function inside a derived class; it's just not needed.) In this program, **base** is inherited by both **derived1** and **derived2**. Inside each class definition, **vfunc( )** is redefined relative to that class. Inside **main( )**, four variables are declared:

Name	**Type**
p	base class pointer
b	object of base
d1	object of derived1
d2	object of derived2

Next, **p** is assigned the address of **b,** and **vfunc( )** is called via **p**. Since **p** is pointing to an object of type **base**, that version of **vfunc( )** is executed. Next, **p** is set to the address of **d1**, and again **vfunc( )** is called by using **p**. This time **p** points to an object of type **derived1**. This causes **derived1::vfunc( )** to be executed. Finally, **p** is assigned the address of **d2**, and **p >vfunc( )** causes the version of **vfunc( )** redefined inside **derived2** to be executed. The key point here is that the kind of object to which **p** points determines which version of **vfunc( )** is executed. Further, this determination is made at run time, and this process forms the basis for run-time polymorphism. Although you can call a virtual function in the "normal" manner by using an object's name and the dot operator, it is only when access is through a base-class pointer (or reference) that run-time polymorphism is achieved.

For example, assuming the preceding example, this is syntactically valid:

d2.vfunc(); // calls derived2's vfunc()

Although calling a virtual function in this manner is not wrong, it simply does not take advantage of the virtual nature of **vfunc( )**.

At first glance, the redefinition of a virtual function by a derived class appears similar to function overloading. However, this is not the case, and the term *overloading* is not applied to virtual function redefinition because several differences exist. Perhaps the most important is that the prototype for a redefined virtual function must match exactly the prototype specified in the base class. This differs from overloading a normal function, in which return types and the number and type of parameters may differ. (In fact, when you overload a function, either the number or the type of the parameters *must* differ! It is through these differences that C++ can

select the correct version of an overloaded function.) However, when a virtual function is redefined, all aspects of its prototype must be the same. If you change the prototype when you attempt to redefine a virtual function, the function will simply be considered overloaded by the C++ compiler, and its virtual nature will be lost.

Another important restriction is that virtual functions must be nonstatic members of the classes of which they are part. They cannot be **friend**s. Finally, constructor functions cannot be virtual, but destructor functions can. Because of the restrictions and differences between function overloading and virtual function redefinition, the term *overriding* is used to describe virtual function redefinition by a derived class.

Source : http://elearningatria.files.wordpress.com/2013/10/cse-iii-object-oriented-programming-with-c-10cs36-notes.pdf