Virtual Function in C++

A virtual function a member function which is declared within base class and is re-defined (Overriden) by derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a **virtual** keyword in base class.
- The resolving of function call is done at Run-time.

Rules for Virtual Functions

1. They Must be declared in public section of class.
2. Virtual functions cannot be static and also cannot be a friend function of another class.
3. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
4. The prototype of virtual functions should be same in base as well as derived class.
5. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
6. A class may have virtual destructor but it cannot have a virtual constructor.

Compile-time(early binding) VS run-time(late binding) behavior of Virtual Functions

Consider the following simple program showing run-time behavior of virtual functions.

```cpp
// CPP program to illustrate
// concept of Virtual Functions
#include<iostream>
using namespace std;

class base
{
public:
    virtual void print ()
    { cout<< "print base class" <<endl; }

    void show ()
    { cout<< "show base class" <<endl; }
};

class derived:public base
{
public:
    void print ()
    { cout<< "print derived class" <<endl; }

    void show ()
    { cout<< "show derived class" <<endl; }
};

int main()
{
    base *bptr;
    derived d;
    bptr = &d;

    //virtual function, binded at runtime
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();
}
```

Output:

```
print derived class
show base class
```

Explanation: Runtime polymorphism is achieved only through a pointer (or reference) of base class type. Also, a base class pointer can point to the objects of base class as well as to the objects of derived class. In above code, base class pointer 'bptr' contains the address of object 'd' of derived class.

Late binding(Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer) an Early binding(Compile time) is done according to the type of pointer, since print() function is declared with virtual keyword so it will be binded at run-time (output is *print derived class* as pointer is pointing to object of derived class) and show() is non-virtual so it will be binded during compile time(output is *show base class* as pointer is of base type).

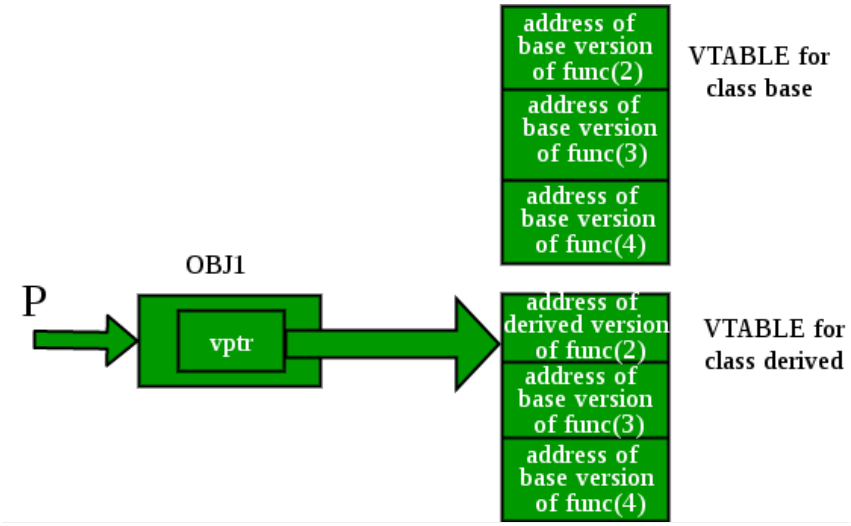
NOTE: If we have created virtual function in base class and it is being overrided in derived class then we don't need virtual keyword in derived class, functions are automatically considered as virtual functions in derived class.

Working of virtual functions(concept of VTABLE and VPTR)

As discussed here , If a class contains a virtual function then compiler itself does two things:

1. If object of that class is created then a **virtual pointer(VPTR)** is inserted as a data member of the class to point to VTABLE of that class. For each new object created, a new virtual pointer is inserted as a data member of that class.
2. Irrespective of object is created or not, **a static array of function pointer called VTABLE** where each cell contains the address of each virtual function contained in that class.

Consider the example below:




```cpp
// CPP program to illustrate
// working of Virtual Functions
#include<iostream>
using namespace std;

class base
{
public:
    void fun_1() { cout << "base-1\n"; }
    virtual void fun_2() { cout << "base-2\n"; }
    virtual void fun_3() { cout << "base-3\n"; }
    virtual void fun_4() { cout << "base-4\n"; }
};

class derived : public base
{
public:
    void fun_1() { cout << "derived-1\n"; }
    void fun_2() { cout << "derived-2\n"; }
    void fun_4(int x) { cout << "derived-4\n"; }
};

int main()
{
    base *p;
    derived obj1;
    p = &obj1;

    // Early binding because fun1() is non-virtual
    // in base
    p->fun_1();

    // Late binding (RTP)
    p->fun_2();

    // Late binding (RTP)
    p->fun_3();

    // Late binding (RTP)
    p->fun_4();

    // Early binding but this function call is
    // illegal(produces error) becasue pointer
    // is of base type and function is of
    // derived class
    //p->fun_4(5);
}
```

Output:

```
base-1
derived-2
base-3
base-4
```

Explanation: Initially, we create pointer of type base class and initialize it with the address of derived class object. When we create an object of derived class, compiler creates a pointer as a data member of class containing the address of VTABLE of derived class.

Similar concept of **Late and Early Binding** is used as in above example. For fun_1() function call, base class version of function is called, fun_2() is overrided in derived class so derived class version is called, fun_3() is not overrided in derived class and is virtual function so base class version is called, similarly fun_4() is not overrided so base class version is called.

NOTE: fun_4(int) in derived class is different from virtual function fun_4() in base class as prototype of both the function is different.

This article is contributed by **Yash Singla**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Recommended Posts:

What happens when a virtual function is called inside a non-virtual function in C++
Default arguments and virtual function
C++ interview questions on virtual function and abstract class
Virtual Destructor
Pure virtual destructor in C++
Can virtual functions be inlined?
Can virtual functions be private in C++?
Virtual destruction using shared_ptr in C++
Virtual base class in C++
Can static functions be virtual in C++?
Advanced C++ | Virtual Constructor
Advanced C++ | Virtual Copy Constructor
Virtual functions in derived classes
Calling virtual methods in constructor/destructor in C++
Pure Virtual Functions and Abstract Classes in C++

Improved By : Prashant Upadhyay 2

Article Tags : C++ cpp-inheritance cpp-virtual

Practice Tags : CPP

Previous

strcpy in C/C++

Next

Calling virtual methods in constructor/destructor in C++

Most popular in C++

- How to create a List with Constructor in C++ STL
- Compiling with g++
- Finding Floor and Ceil of a Sorted Array using C++ STL
- Zero Initialization in C++
- C++ Mathematical Functions

More related articles in C++

- Generate a random permutation of elements from range [L, R] (Divide and Conquer)
- Can C++ reference member be declared without being initialized with declaration?
- How to find the minimum and maximum element of a Vector using STL in C++?

Internal details of std::sort() in C++

Generate all the binary strings of N bits