# Virtual Function in C++

A virtual function (also known as virtual methods) is a member function that is declared within a base class and is re-defined (overridden) by a 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 method.

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

## Rules for Virtual Functions

The rules for the virtual functions in C++ are as follows:

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

### Compile time (early binding) VS runtime (late binding) behavior of Virtual Functions

Consider the following simple program showing the runtime behavior of virtual functions.

- C++

```cpp
// C++ program to illustrate
// concept of Virtual Functions

#include <iostream>
using namespace std;

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

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

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

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

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

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

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

    return 0;
}
```

**Output**

```
print derived class

show base class
```

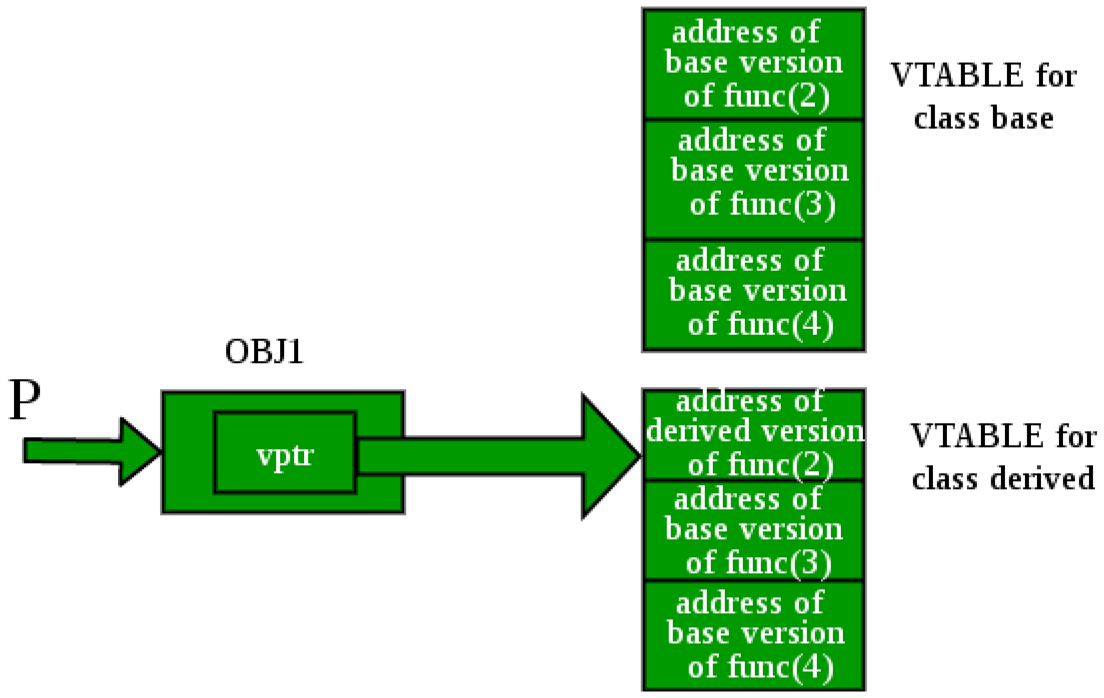
**Explanation:** Runtime polymorphism is achieved only through a pointer (or reference) of the base class type. Also, a base class pointer can point to the objects of the base class as well as to the objects of the derived class. In the above code, the base class pointer 'bptr' contains the address of object 'd' of the derived class.
Late binding (Runtime) is done in accordance with the content of the pointer (i.e. location pointed to by pointer) and Early binding (Compile-time) is done according to the type of pointer since the print() function is declared with the virtual keyword so it will be bound at runtime (output is *print derived class* as the pointer is pointing to object of derived class) and show() is non-virtual so it will be bound during compile time (output is *show base class* as the pointer is of base type).
***Note:** If we have created a virtual function in the base class and it is being overridden in the derived class then we don't need a virtual keyword in the derived class, functions are automatically considered virtual functions in the derived class.*

## Working of Virtual Functions (concept of VTABLE and VPTR)

As discussed here, if a class contains a virtual function then the compiler itself does two things.

1. If an object of that class is created then a **virtual pointer (VPTR)** is inserted as a data member of the class to point to the 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 whether the object is created or not, the class contains as a member **a static array of function pointers called VTABLE**. Cells of this table store the address of each virtual function contained in that class.

Consider the example below:


address of base version of func(2)
address of base version of func(3)
address of base version of func(4)
VTABLE for class base
OBJ1
P
vptr
address of derived version of func(2)
address of base version of func(3)
address of base version of func(4)
VTABLE for class derived

- C++

```
// C++ 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) because pointer
    // is of base type and function is of
    // derived class
    // p->fun_4(5);

    return 0;
}
```

**Output**

```
base-1

derived-2

base-3

base-4
```

**Explanation:** Initially, we create a pointer of the type base class and initialize it with the address of the derived class object. When we create an object of the derived class, the compiler creates a pointer as a data member of the class containing the address of VTABLE of the derived class.
A similar concept of **Late and Early Binding** is used as in the above example. For the fun_1() function call, the base class version of the function is called, fun_2() is

overridden in the derived class so the derived class version is called, fun_3() is not overridden in the derived class and is a virtual function so the base class version is called, similarly fun_4() is not overridden so base class version is called.
***Note:*** *fun_4(int) in the derived class is different from the virtual function fun_4() in the base class as prototypes of both functions are different.*

## Limitations of Virtual Functions

- **Slower:** The function call takes slightly longer due to the virtual mechanism and makes it more difficult for the compiler to optimize because it does not know exactly which function is going to be called at compile time.
- **Difficult to Debug:** In a complex system, virtual functions can make it a little more difficult to figure out where a function is being called from.