**Virtual Function in C++**

A virtual function is a member function which 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 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](https://www.geeksforgeeks.org/polymorphism-in-c/)
* Functions are declared with a **virtual**keyword in base class.
* The resolving of function call is done at runtime.

**Rules for Virtual Functions**

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 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 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 [virtual destructor](https://www.geeksforgeeks.org/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 runtime behavior of virtual functions.

// CPP 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 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) and Early binding (Compile time) is done according to the type of pointer, since print() function is declared with virtual keyword so it will be bound at runtime (output is *print derived class* as 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 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 virtual keyword in the derived class, functions are automatically considered as virtual functions in the derived class.

**Working of virtual functions (concept of VTABLE and VPTR)**  
As discussed [here](https://www.geeksforgeeks.org/virtual-functions-and-runtime-polymorphism-in-c-set-1-introduction/), 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, 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: 



|  |
| --- |
| // 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) 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 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.

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 overridden in derived class so derived class version is called, fun\_3() is not overridden in derived class and is virtual function so base class version is called, similarly fun\_4() is not overridden so base class version is called.

**NOTE:** fun\_4(int) in derived class is different from virtual function fun\_4() in base class as prototypes of both the functions are different.