Virtual functions in C++: Runtime Polymorphism

we will see **what are virtual functions and why we use them**.

When we declare a function as virtual in a class, all the sub classes that override this function have their function implementation as virtual by default (whether they mark them virtual or not).

Why we declare a function virtual?

A virtual function is a member function within the base class that we redefine in a derived class. It is declared using the virtual keyword. When a class containing virtual function is inherited, the derived class redefines the virtual function to suit its own needs.

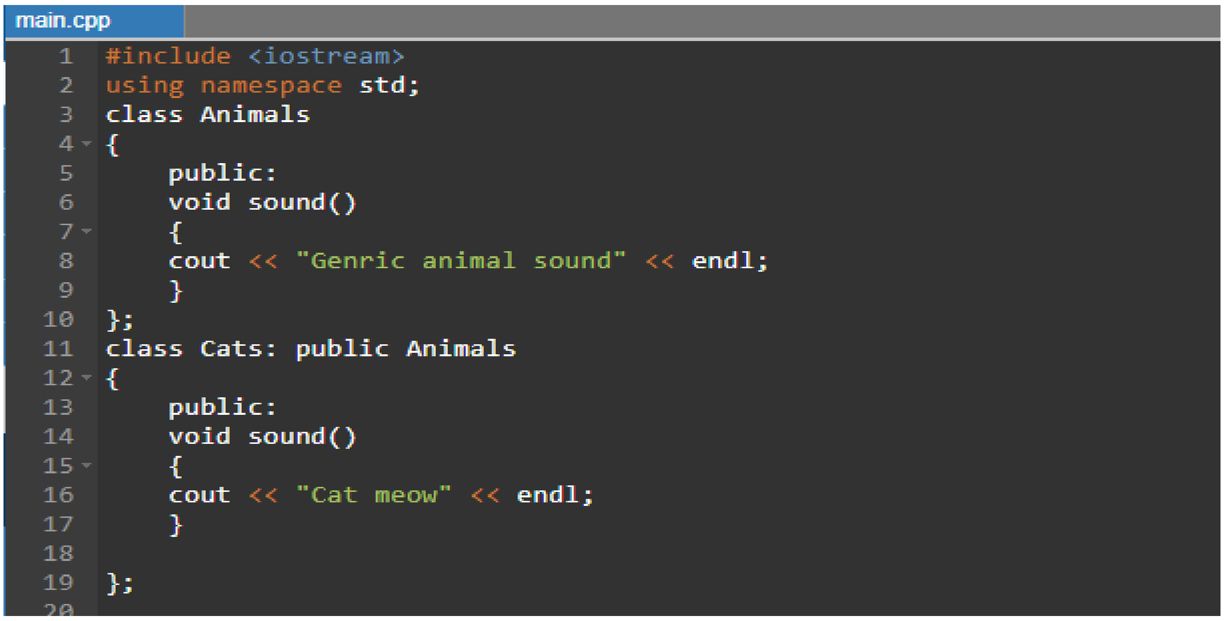
Rules for Virtual Function in C++:

1. They are always defined in a base class and overridden in derived class but it is not mandatory to override in the derived class.
2. The virtual functions must be declared in the public section of the class.
3. They cannot be static or friend function also cannot be the virtual function of another class.
4. The virtual functions should be accessed using a pointer to achieve run time polymorphism.

Early Binding

Early binding is a phenomenon wherein the decision to match various function calls happens at the compile time itself and the compiler directly associates the link with addresses. It is also known as Static Binding or Compile-time Binding.

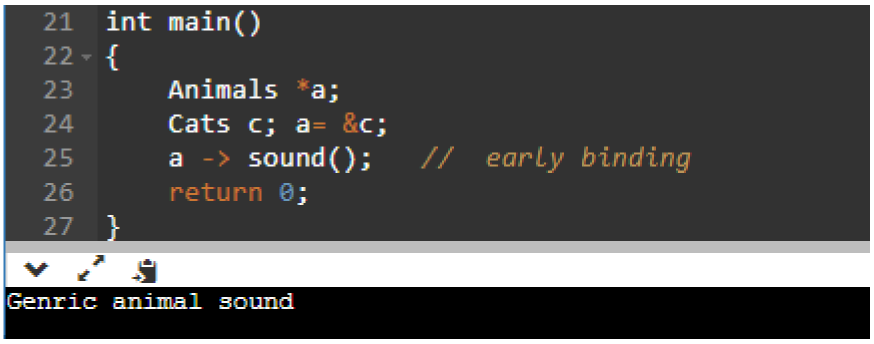
1. As we know we write code in the high-level language
2. Then the compiler converts this into low-level language that computer can understand, mostly machine language at the time of compilation
3. In early binding, the compiler directly provides the address of function declaration instruction to the function call instruction
4. Thus as the name suggests the binding happens very early before the program runs.



main.cpp

```cpp
#include <iostream>
using namespace std;
class Animals
{
	public:
	void sound()
	{
	cout << "Genric animal sound" << endl;
	}
};
class Cats: public Animals
{
	public:
	void sound()
	{
	cout << "Cat meow" << endl;
	}

};
```



```cpp
int main()
{
	Animals *a;
	Cats c; a= &c;
	a -> sound();   //  early binding
	return 0;
}
```

```
Genric animal sound
```

Explanation

In this example, we created a pointer a to the parent class Animals. Then by writing a= &c, the pointer 'a' started referring to the object c of the class Cats.
a -> sound(); – On calling the function sound() which is present in both the classes by the pointer 'a', the function of the parent class got called, even if the pointer is referring to the object of the class Cats.

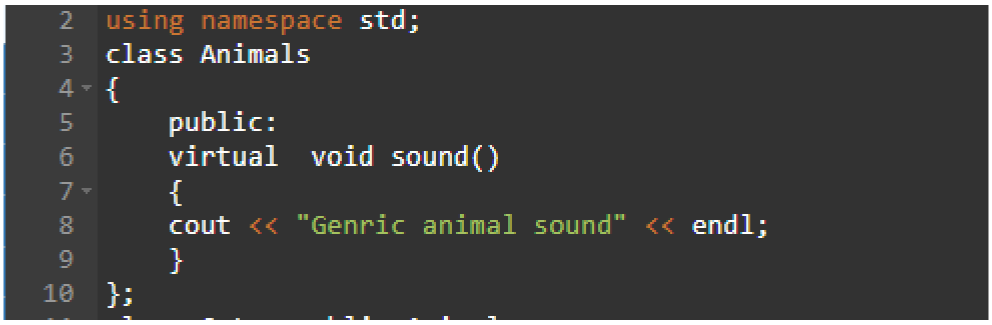
This is due to Early Binding. We know that 'a' is a pointer of the parent class referring to the object of the child class. Since early binding takes place at compile-time, therefore when the compiler saw that 'a' is a pointer of the parent class, it matched the call with the 'sound()' function of the parent class without searching for object the pointer it is referring to.

Moving on with this article on Virtual Function in C++

Late Binding

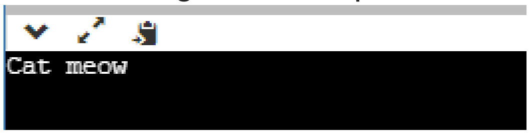
In late binding, the compiler identifies the object at runtime and then matches the function call with the correct function. It is also known as Dynamic Binding or Runtime Binding.

Late binding in the above problem may be solved by using virtual keyword in the base class. Let's see how this happens by using the above example itself, but only adding virtual keyword.



```
using namespace std;
class Animals
{
    public:
    virtual  void sound()
    {
    cout << "Genric animal sound" << endl;
    }
};
```

After Adding virtual output will changed



```
Cat meow
```

Explanation

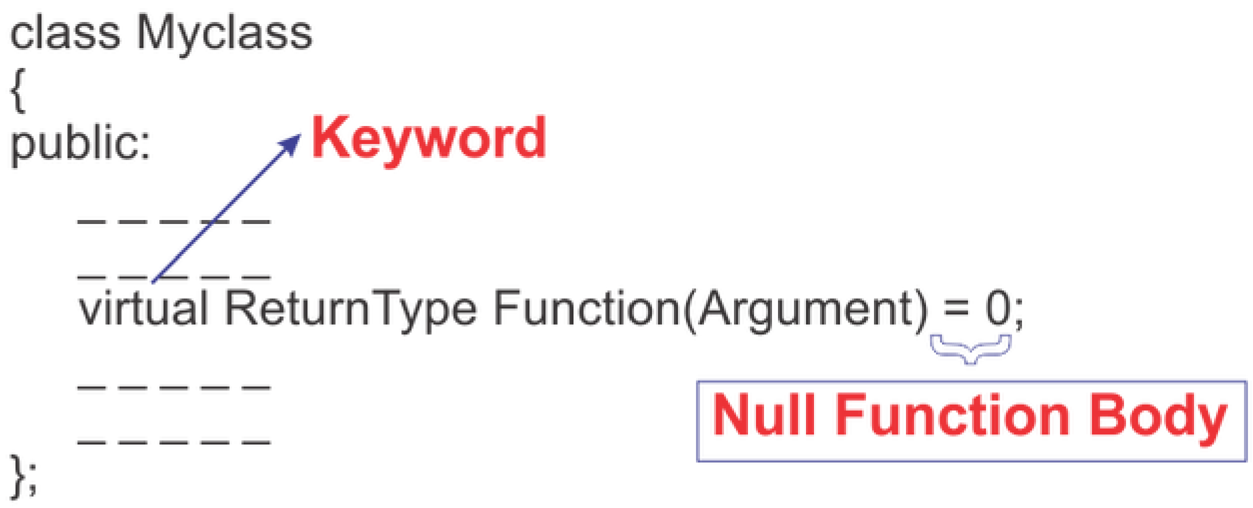
Here the function sound() of the base class is made virtual, thus the compiler now performs late binding for this function. Now, the function call of the sound() function will be matched to the function definition at runtime. Since the compiler now identifies pointer ‘a’ as referring to the object ‘c’ of the derived class Cats, it will call the sound() function of the class Cats.

Pure Virtual Function

Virtual function define inside the base class normally serve as a frame work for future design of the hierarchy, these function can be overridden by the method in the derived classes. In most of the cases, these virtual function are defined with null body, it has no definition. Such function in the base class are similar to do-nothing or dummy function and in C++ they are called pure virtual function. Pure virtual function declared as a virtual function with its declaration followed by =0.

A pure virtual function declared in a base class has no implementation as for as the base class is conserved. The classes derived from a base class having a pure virtual function have to define such a function or re-declare it as a pure virtual function. It must be noted that, a class containing pure virtual function cannot be used to define any objects of its own and hence such classes are called pure abstract class or simply abstract class. Whereas all other classes without pure virtual function and which are initiated are called concrete class. A pure virtual function is an undefined place holder that the derived class is accepted to complete. The following are the properties of pure virtual function-

1. A pure virtual function has no implementation in the base class hence, a class with pure virtual function cannot be initiated.
2. It acts as an empty bucket that the derived class is supposed to fill.
3. A pure virtual member function cannot be invoked by its derived class.



```
class Myclass
{
public:
    _ _ _ _ _ _
    _ _ _ _ _ _
    virtual ReturnType Function(Argument) = 0;
    _ _ _ _ _ _
    _ _ _ _ _ _
};
```

Needs for virtual function: – The following rule hold good with respect to virtual function.

1. When a virtual function in a base class is a created, there must be definition of a virtual function in the base class even if base class version of the function is never actually called. However pure virtual function are exception.
2. They cannot be static member.
3. They can be friend function to another class.
4. They are accessed using object pointer.
5. A base pointer can serve as a pointer to derived object since it is type compatible where as a derived object pointer variable to base object.
6. Its prototype in a base class and derived class must be identical for the virtual function to work properly.
7. The class cannot have virtual constructor, but can content virtual destructor. In fact, virtual destructor are essential to the solution of some problem. It is also possible to have virtual operator overloading.
8. More importantly to realize the potential benefit of virtual function supporting run time polymorphism, they should be declared in the public section of class.