Importance of virtual function, function call binding, virtual functions, implementing late binding, need for virtual functions, abstract base classes and pure virtual functions, virtual destructors

---

Importance of virtual function: A virtual function is a member function that is declared within a base class and redefined by a derived class. To create virtual function, precede the function's declaration in the base class with the keyword virtual. When a class containing virtual function is inherited, the derived class redefines the virtual function to suit its own needs.

Virtual Functions are used to support "Run time Polymorphism". You can make a function virtual by preceding its declaration within the class by keyword 'virtual'. When a Base Class has a virtual member function, any class that inherits from the Base Class can redefine the function with exactly the same prototype i.e. only functionality can be redefined, not the interface of the function. A Base class pointer can be used to point to Base Class Object as well as Derived Class Object. When the virtual function is called by using a Base Class Pointer, the Compiler decides at Runtime which version of the function i.e. Base Class version or the overridden Derived Class version is to be called. This is called Run time Polymorphism.

What is Virtual Function?

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.

Virtual Function is a member function of the base class which is overridden in the derived class.
The compiler performs late binding on this function.

To make a function virtual, we write the keyword virtual before the function definition.
A virtual function is a member function that you expect to be redefined in derived classes. 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 expression used to make the function call. Functions in derived classes override virtual functions in base classes only if their type is the same.

Rules for Virtual Function in C++

1. Virtual functions must be members of some class.
2. Virtual functions cannot be static members.
3. They are accessed through object pointers.
4. They can be a friend of another class.
5. A virtual function must be defined in the base class, even though it is not used..

What is binding?

Binding for functions means that wherever there is a function call, the compiler needs to know which function definition it should be matched to. This depends upon the signature of each

function declaration & the assignments that are taken. Also, the compiler needs to know that when this matching between the function call and choosing the correct definition will happen.

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.

Then the compiler converts this into low-level language that computer can understand, mostly machine language at the time of compilation

In early binding, the compiler directly provides the address of function declaration instruction to the function call instruction

Thus as the name suggests the binding happens very early before the program runs.

Example

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

Output: 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.

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.

Example

```
#include <iostream>
using namespace std;
class Animals
{public:
virtual void sound()
{
cout << "Genric animal sound" << endl;
}
};
class Cats: public Animals
{
public:
void sound()
{
cout << "Cats meow" << endl; } };
int main()
  {
        Animals *a;
        Cats c; a= &c; a -> sound();
        return 0;
}
```

Output: Cats 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

A pure virtual function in C++ is a virtual function for which we don't have an implementation, we only declare it. A pure virtual function is declared by assigning 0 in the declaration.

virtual void sound() = 0;

Here sound() is a pure virtual fuction.

Abstract Class

An abstract class is defined as a class with one or more pure virtual functions. As explained above pure virtual function is a virtual member function that is marked as having no implementation. It has no implementation possible with the information provided in the class, including any base classes. An abstract class is also known as an abstract base class.

Example

```
#include <iostream>
using namespace std;
class Employee                          //  abstract base class
{
virtual int getSalary() = 0;            // pure virtual function
};
class Employee_1: public Employee
{
int salary;
public:
Employee_1(int s)
{
salary = s;
}
int getSalary()
{
return salary;
}
};
class Employee_2: public Employee
{
int salary;
public:
Employee_2(int t)
{
salary = t;
}
int getSalary()
```

```
{
return salary;
}
};
int main()
{
Employee_1 e1(5000);
Employee_2 e2(3000);
int a, b;
a = e1.getSalary();
b = e2.getSalary();
cout << "Salary of Developer : " << a << endl;
cout << "Salary of Driver : " << b << endl;
return 0;
}
```

Output:- Salary of Developer : 5000
Salary of Driver : 3000

Explanation

The 'getSalary()' function in the class Employee is a pure virtual function. Since the Employee class contains the pure virtual function, therefore it is an abstract base class.
Since the pure virtual function is defined in the subclasses, therefore the function 'getSalary()' is defined in both the subclasses of the class Employee i.e in Employee_1 and Employee_2.

Example for Virtual Function

```
#include<iostream>
using namespace std;
class base
{
public:
void function_1() {
        cout << "base class function 1\n"; }
virtual void function_2()
 {
 cout << "base class function 2\n"; }
virtual void function_3()
{
cout << "base class function 3\n";
}
virtual void function_4()
{
cout << "base class function 4\n";
```

```
}
};
class derived : public base
{
public:
void function_1() {
cout << "derived class function 1\n";
}
void function_2()
{
cout << "derived class function 2\n"; }
void function_4(int x) {
 cout << "derived class function 4\n"; }
};
int main() {
            base *ptr;
          derived obj1;
          ptr = &obj1;
          ptr->function_1();
          ptr->function_2();
          ptr->function_3();
          ptr->function_4();

   return 0;
}
```

Output- base class function 1
derived class function 2
base class function 3
base class function 4

Explanation

For function_1() function call, base class version of function is called, function_2() is overridden in derived class so derived class version is called, function_3() is not overridden in derived class and is virtual function so base class version is called, similarly function_4() is not overridden so base class version is called.

Virtual Destructor

Deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor.

Although C++ provides a default destructor for your classes if you do not provide one yourself, it is sometimes the case that you will want to provide your own destructor (particularly if the class needs to deallocate memory). You should **always** make your destructors virtual if you're dealing with inheritance.