Virtual Functions & Polymorphism

Run-time Polymorphism

Consider a situation where we have Derived Class which has overridden some method of the Base Class. Polymorphism allows us to have a Base Class reference a Derived Class Object. Then, in such a case, which function to call (Base or Derived) is decided at run-time, as the compiler is unable to resolve which one to call during compilation. Below is a classic example of Run-time Polymorphism:

```
#include <bits/stdc++.h>
using namespace std;

class Base
{
    public:
        void whoami() {
            cout << "I'm Base\n";
        }
};

class Derived: public Base
{
    public:
        void whoami() {
            cout << "I'm Derived\n";
        }
};

int main(void)
{
    Base *b_ptr = new Derived;

    //run-time polymorphism
    b_ptr->whoami();
```

```
    return 0;
}
```

Output:

```
I'm Base
```

In the above code, since, the pointer reference is of Base-type, so at runtime, it is resolved to call the *whoami( )* version of the Base class. Therefore, *"I'm Base"* is printed. But, suppose we want to call the Derived Class's version of the function. Here, comes the play of **virtual** keyword.

Virtual Functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object. For example, consider a employee management software for an organization, let the code has a simple base class *Employee* , the class contains virtual functions like *raiseSalary( ), transfer( ) , promote( ),..* etc. Different types of employees like *Manager* , *Engineer* , ..etc may have their own implementations of the virtual functions present in base class *Employee*. In our complete software, we just need to pass a list of employees everywhere and call appropriate functions without even knowing the type of employee. e.g. we can easily raise the salary of all employees by iterating through the list of employees. Every type of employee may have its own logic in its class, we don't need to worry because if *raiseSalary( )* is present for a specific employee type, only that function would be called.

```
class Employee
{
    public:
        virtual void raiseSalary()
        {  /* common raise salary code */  }

        virtual void promote()
        { /* common promote code */ }
};

class Manager: public Employee
{
    public:
```

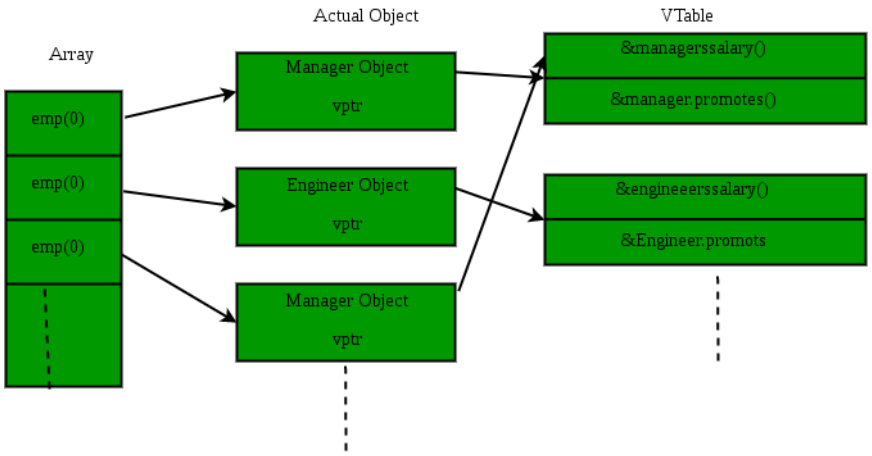
```
        virtual void raiseSalary()
        {  /* Manager specific raise salary code, may contain
              increment of manager-specific incentives*/  }

        virtual void promote()
        { /* Manager specific promote */ }
};

// Similarly, there may be other types of employees

// We need a very simple function to increment the salary of
// Note that emp[] is an array of pointers and actual pointed
// be any type of employees. This function should ideally be
// like Organization, we have made it global to keep things s
void globalRaiseSalary(Employee *emp[], int n)
{
    for (int i = 0; i < n; i++)
        emp[i]->raiseSalary(); // Polymorphic Call: Calls rai
                               // according to the actual obj
                               // according to the type of po
}
```

like *globalRaiseSalary( )*, there can be many other operations that can be appropriately done on a list of employees without even knowing the type of actual object.

Virtual functions are so useful that later languages like Java kept all the methods as virtual by default.

How does compiler do this magic of late resolution?

Compiler maintains two things to this magic :



vtable - A table of function pointers. It is maintained per class.

vptr - A pointer to vtable. It is maintained per object.

Compiler adds additional code at two places to maintain and use

vptr.

1. Code in every constructor. This code sets vptr of the object being created. This code sets *vptr* to point to *vtable* of the class.
2. Code with polymorphic function call (e.g. *bp->show( )* in above code). Wherever a polymorphic call is made, compiler inserts code to first look for *vptr* using base class pointer or reference (In the above example, since pointed or referred object is of derived type, vptr of derived class is accessed). Once *vptr* is fetched, *vtable* of derived class can be accessed. Using *vtable* , address of derived derived class function *show()* is accessed and called.