**Virtual Functions and Runtime Polymorphism in C++ | Set 1 (Introduction)**

Consider the following simple program which is an example of runtime polymorphism.  
The main thing to note about the program is, derived class function is called using a base class pointer. The idea is, [virtual functions](http://www.geeksforgeeks.org/virtual-function-cpp/) are called according to the type of object pointed or referred, not according to the type of pointer or reference. In other words, virtual functions are resolved late, at runtime.

|  |
| --- |
| #include<iostream>  using namespace std;    class Base  {  public:      virtual void show() { cout<<" In Base \n"; }  };    class Derived: public Base  {  public:      void show() { cout<<"In Derived \n"; }  };    int main(void)  {      Base \*bp = new Derived;      bp->show();  // RUN-TIME POLYMORPHISM      return 0;  } |

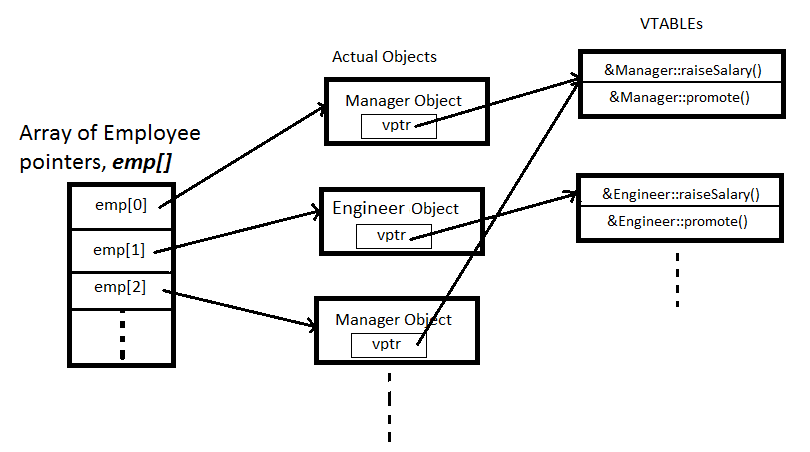
Output:

In Derived

**What is the use?**  
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. For example, we can easily raise salary of all employees by iterating through 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 {      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 salary of all employees  // Note that emp[] is an array of pointers and actual pointed objects can  // be any type of employees. This function should ideally be in a class  // like Organization, we have made it global to keep things simple  void globalRaiseSalary(Employee \*emp[], int n)  {      for (int i = 0; i < n; i++)          emp[i]->raiseSalary(); // Polymorphic Call: Calls raiseSalary()                                 // according to the actual object, not                                 // according to the type of pointer  } |

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 keep all methods as virtual by default](http://www.geeksforgeeks.org/g-fact-43/).

**How does compiler do this magic of late resolution?**  
Compiler maintains two things to this magic:  
[](http://www.geeksforgeeks.org/wp-content/uploads/virtualFuns1.png)  
[***vtable:***](http://en.wikipedia.org/wiki/Virtual_method_table) A table of function pointers. It is maintained per class.  
[***vptr:***](http://en.wikipedia.org/wiki/Virtual_method_table#Implementation) A pointer to vtable. It is maintained per object (See [this](http://geeksquiz.com/c-virtual-functions-question-12/) for an example).

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.

**Is this a standard way for implementation of run-time polymorphism in C++?**  
The C++ standards do not mandate exactly how runtime polymophism must be implemented, but compilers generally use minor variations on the same basic model.