Pointers, Virtual Functions and Polymorphism

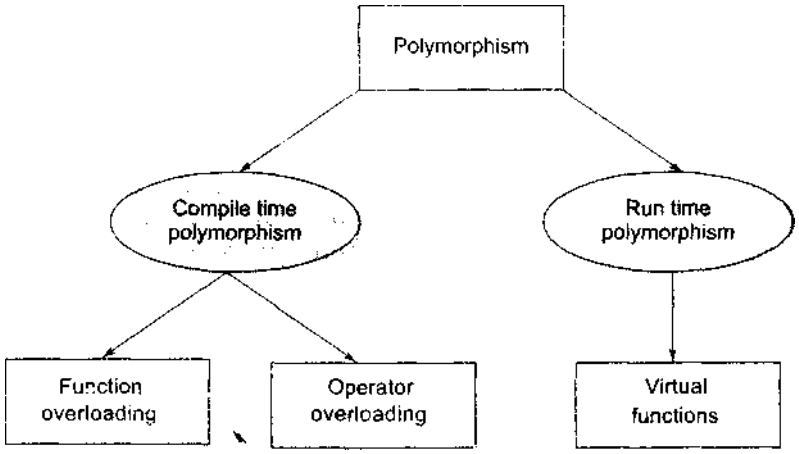
Polymorphism is one of the crucial features of OOP. It simply means **'one name, multiple forms'**. We have already seen how the concept of ***polymorphism*** is implemented using the overloaded functions and operators. The overloaded member functions are 'selected' for invoking matching arguments, both type and number. This information is known to the compiler at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called ***early binding*** or ***static binding*** or ***static linking.*** Also known as compile time polymorphism, early binding simply means that an object is bound to its function call at compile time.

Now let us consider a situation where the function name and prototype is the same in base and derived classes. For example, consider the following class definitions:

```
Class A
{
  int x;
  public :

  void show ( );
  {
          … …
  }
};

class B : public A
{
  int y;
  public :

  void show ( )
  {
          … …
  }
};
```

How do we use the member function show ( ) to print the values of objects of both classes A and B? Since the prototype of show ( ) is the same in both the places, the function is not overloaded and therefore static binding does not apply. We have seen earlier that in such situations, we may use the class resolution operator to specify the class while invoking the functions with the derived class objects.

It would be nice if the appropriate member function could be selected while the program is running. This is known as run time polymorphism. How could it happen? C++ supports a mechanism known as virtual function to achieve run time polymorphism. Please refer following Fig.



At run time, when it is known what class objects are under consideration, the appropriate version of the function is invoked. Since the function is linked with a particular class much later after the compilation, this process is termed as late binding. It is also known as dynamic binding because the selection of the appropriate function is done dynamically at run time.

Dynamic binding is one of the powerful features of C++. This requires the use of pointers to objects. We shall discuss in detail how the object pointers and virtual function are used to implement dynamic binding

Pointers

A pointer is one of the key aspects of C++ language similar to that of C. As we know, pointers offer a unique approach to handle data in C and C++. We have seen some of the applications of pointers early. In this section, we shall discuss the rudiments of pointers and special usage of them in C++.

We know that a pointer is a derived data type that refers to another data variable by storing the variable's memory address rather than data. A pointer variable defines where to get the value of a specific data variable instead of defining actual data.

Like C, a pointer variable can also refer to (or point to) another pointer in C++. However, it often points to a data variable. Pointers provide an alternative approach to access other data objects.

Declaring and Initializing Pointers

We can declare a pointer variable similar to other variables in C++. Like C, the declaration is based on the data type of the variable it points to. The declaration of a pointer variable takes the following form:

data-type *pointer-variable;

Here pointer-variable is the name of the pointer, and the data-type refers to one of the valid C++ data types, such as int, char, float, and so on. The data-type is followed by an asterisk (*) symbol, which distinguishes a pointer variable from other variables to the compiler.

We can locate asterisk (*) immediately before the pointer variable, or between the data type and the pointer variable, or immediately after the data type. It does not cause any effect in the execution process.

As we know, a pointer variable can point to any type of data available in C++. However, it is necessary to understand that a pointer is able to point to only one data type at the specific time. Let us declare a pointer variable, which points to an integer variable, as follows:

int *ptr;

ptr is a pointer variable and points to an integer data type. The pointer variable could contain the memory location of any integer variable. In the same manner, we can declare pointer variables for other data types also.

Pointers to Objects

We have already seen how to use pointers to access the class members. As stated earlier, a pointer can point to an object created by a class. Consider the following statement:

item x;

where item is a class and x is an object defined to be of type item. Similarly we can define a pointer it_ptr of type item as follows: item *it_ptr;
Object pointers are useful in creating objects at run time. We can also use an object pointer to access the public members of an object. Consider a class item defined as follows:

```
Class item
{
        int code;
        float prize;
        public:
                void getdata (int a, int b)
                {
                        code = a;
                        prize = b;
                }
                void show( )
                {
                        cout << “code : “ << code << “\n”;
                        cout << “Prize : “ << prize << “\n;
                }
};

void main ( )
{
  item x;
  item *ptr = &x;

  ptr -> getdata (111, 100);
  ptr -> show ( );

  ptr = new item (222, 200);

  ptr -> getdata (111, 100);
  ptr -> show ( );
}
```

this Pointer

C++ uses a unique keyword called this to represent an object that invokes a member function. this is a pointer that points to the object for which this function was called. For example the function call A.max ( ) will set the pointer this to the address of the object A. The starting address is the same as the address of the first variable in the class structure.

This unique pointer is automatically passed to a member function when it is called. The pointer this acts as an implicit argument to all the member functions. Consider the following simple example:

```
class ABC
{
  int a;
  ... ...
  ... ...
};
```

The private variable 'a' can be used directly inside a member function, like a = 123; we can also use the following statement to do the same job:

```
this -> a =  123;
```

Since C++ permits the use of shorthand form a = 123, we have not been using the pointer this explicitly so far. However, we have been implicitly using the pointer this when overloading the operators using member function.

Recall that, when, a binary operator is overloaded using a member function, we pass only one argument to the function. The other argument is implicitly passed using the pointer this. one important application of the pointer this is to return the object it points to. For example the statement

```
return  *this;
```

inside a member function will return the object that invoked the function. This statement assumes importance when we want to compare two or more objects inside a member function and return the invoking object as a result. Remember, the dereference operator * produces the contents at the address contained in the pointer.

Virtual Functions

As mentioned earlier, polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different forms. An essential requirement of polymorphism is therefore the ability to refer to objects without any regard to their classes. This necessitates the use of a single pointer variable to refer to the objects of different classes. Here, we use the pointer to base class to refer to all the derived objects.

But we just discovered that a base pointer, even when it is made to contain the address of derived class, always executes the function in the base class. The compiler simply ignores the contents of the pointer and chooses the member function that matches the type of the pointer. How do we then achieve polymorphism? It is achieved using what is known as virtual functions.

When we use the same function name in both the base and derived classes, the function in base class is declared as virtual using the keyword virtual preceding its normal declaration. When a function is made virtual, C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. Thus, by making the base pointer to point to different objects, we ca different versions of the virtual function. Following Program illustrates this point.

```
#include<iostream.h>

class Base
{
  public :
        void display( )
        {
              cout << “Display Base \n”;
        }
        virtual void show ( )
        {
              cout << “ Show Base \n”;
        }
};

class Derive : public Base
{

        void display( )
        {
              cout << “Display Derived \n”;
        }

        void show ( )
        {
              cout << “ Show Derived \n”;
        }
};

int main( )
{
        Base B;
        Derived D;
        Base * bptr;

        cout << “bptr points to Base \n”;
```

```
        bptr = &B;

        bptr -> display( );  //      Call Base version
        bptr -> show ( );    //      Call Base version

        cout << "bptr points to Derived \n";
        bptr = &D;

        bptr -> display( );  //      Call Base version
        bptr -> show ( );    //      Call Derived version

        return 0;
}
```

The output of above program would be :

Bptr points to Base

Display Base
Show Base

Bptr points to Derived

Display Base
Display Derived

One important point to remember is that, we must access virtual functions through the use of a pointer declared as a pointer to the base class. Why can't we use the object name with the dot operator the same way as any other member function to call the virtual functions? We can, but remember, run time polymorphism is achieved only when a virtual function is accessed through a pointers to the base class.

Rules for Virtual Functions

When virtual functions are created for implementing late binding, we should observe basic rules that satisfy the compiler requirements

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

6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.

7. We cannot have virtual constructors, but we can have virtual destructors.

8. While a base pointer can point to any type of the derived object, the reverse is not true. That is to say, we cannot use a pointer to a derived class to access an object of the base type.

9. When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to the next object.

10. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

Pure Virtual Functions

It is normal practice to declare a function virtual inside the base class and redefine it in the derived classes. The function inside the base class is seldom used for performing any task. It only serves as a placeholder. For example, if we have not defined any object of class media and therefore the function display( ) in the base class has been defined 'empty'. Such functions are called "do-nothing" functions.

A “do-nothing" function may be defined as follows:

```
virtual void display( ) = 0;
```

Such functions are called pure virtual functions. A pure virtual function is a function declared in a base class that has no definition relative to the base class. In such cases, the compiler requires each derived class to either define the function or redeclare it as a pure virtual function. Remember that a class containing pure virtual functions cannot be used to declare any objects of its own. As stated earlier, such classes are called abstract base classes. The main objective of an abstract base class is to provide some traits to the derived classes and to create a base pointer required for achieving run time polymorphism.

Pointer to derived class

We can use pointers not only to the base objects but also to the object of the derived classes. Pointers to objects of a base class are type compatible to the objects of a derived class.

Therefore a single pointer variable can be made to point to objects belonging to different classes. For example if B is a base class and D is a derived class from B, then a pointer declared as a pointer to B can also be a pointer to D.

```
B base;
D derived;
B *p;
P=&base;
```

We can make pointer to point to the object of D.

```
P=&derived;
```

In a case a members of D has the same name as one of the members of B, then any reference to that members by pointer will always access the base class members.