Subject : Information Technology
Paper : Object Oriented Concepts & Programming
Module : Runtime polymorphism by virtual functions

Introduction

Polymorphism is considered to be one of the biggest advantages of using an object oriented programming. It is the property of the same object to behave differently in a different context when supplied with the same message. There are two ways in which this can be done. Compile time polymorphism which takes place at compile time and Runtime polymorphism which happens at runtime. The operator overloading and function overloading are examples of compile time polymorphism. When we encounter statements shown in 24.1, we can see that the same message (+) is interpreted differently. Look at these definitions of three different types of data and their respective operations. The + operations which we see indicates the same message given to three different pairs of objects of different types; int-int, char-char, department-employee as shown in 24.2.

```
Int i, j;
char c1, c2;
department d1;
employee e1, e2;
```

Figure 23.1 variables to overload

```
i + j
c1 +c2
d1 + e2
```

Figure 23.3 operator overloading

When the functions are overloaded, the same function acts differently when a different set of arguments are passed to them. For example, the same operation can be executed using an overloaded function called Sum. When the function Sum is called with int types an int addition is performed, when complex arguments are provided, real and imaginary parts are added and when an employee is added to the department, he becomes part of the department, by having two different arguments in Sum. In all of above cases, the object behaves differently, i.e. calls a different function or invokes different operator, given the same message (Sum or +), depending on the context (the arguments that we pass), thus it confirms to be a polymorphism. Both of these operations happen at compile time and thus this is called compile time polymorphism.

```
Sum (I,j)
Sum(c1,c2);
Sum (d1,e2);
```

Figure 23.2 function overloading

The C++ language extends this functionality to runtime. It is possible to define, for example, classes in some hierarchy and have virtual functions defined. We can have a pointer pointing to a base class object but can call a function defined for any derived class object and decide that at run time. For example, consider we have classes shown in Fig 24.4.

```
class Shape…
class Rectangle: public Shape
class Square:: public Rectangle
class circle:: public Shape
```

Figure 23.4 class hierarchy

Once we have this hierarchy, we can have code defined as depicted in figure 24.5.

```
Shape *pShape;

pShape  = new (Circle)
pShape->draw();
pShape = new (Square)
pShape->draw();
```

Figure 24.5 calling draw()

We know that it is possible to use a base class pointer to point to a derived class object and thus can call the functions of the class pointed to by that pointer. However, C++ does not provide that facility by default. In above case, if draw() is defined in all classes as a normal function, in both of above cases, the function of the Shape class is only called. If we explicitly want the function to be decided at runtime based on what the pointer is pointing to, we must define that function as a virtual function.

Calling functions differently

A virtual function can be called in two different ways, using the Object.Function () notation as well as (pointer to object) -> function () notation[1]. The object dot member function method invokes the member virtual function in a static form, resolved at compile time. That means the function is replaced at compile time with the subobject jump instruction where the function is loaded at runtime. If the function is inline the code is copied at the place at compile time. Unlike that, the pointer to object invokes the virtual function at runtime. The process of replacing the call with either the jump to the function code or pasting of the code itself (which is known as resolution) happens at run time. That slows down the execution of the program and thus not chosen as a default behavior of a C++ function. If a programmer wants to make sure that the function call is resolved at runtime, he will have to define a pointer to an object and define that function as virtual. Only, in that case, the function call will be resolved at runtime. As mentioned in the footnote-1, instead of a pointer to an object, one can pick up a reference of an object and use Reference.Function() method to invoke the function at runtime.

When a member function is called using Object.Function notation, an interesting transformation takes place. The member function call is replaced by the following code, having an additional argument; as a pointer to the invoking object.

Thus Object.Function() is replaced by Function (class * object);

For example, in the case of the Emp class that we have seen in the 16th module, we can write following code, assuming a function Display() is defined for displaying details of the employee object.

```
Emp tEmp;
```

[1] Unless explicitly stated, a pointer to object means either pointer to object or reference of an object in the discussion that follows.

```
tEmp.Display()
```

//above statement is internally converted to

```
Display(Emp * tEmp)2
```

The Display() has no parameters originally but have one parameter now. If this function has one or more than one parameters, this additional parameter is added in front of all of them. This special parameter, which is basically a pointer to the invoking object, is known as this pointer and a programmer can access that using a keyword *this*.

When can a programmer need to use this pointer? Here is one such example depicted in the program 24.1 block-1. Here is a program which decides who is elder, whether an invoking object or the object which is passed as an argument. The idea is to make sure the function returns the elder person out of the two, either the invoking object or the object which is passed. The return of invoking object is the area of our interest. That object is referred as *this and following code decides what the function returns.

```
// Program 23.1 Block 1
//ThisPointer.cpp
#include <iostream>
#include <string>
using namespace std;

class Hero
{
   string nm;
   int Age;
public:
   Hero (string t_nm, int t_Age)
   {
      nm = t_nm;
      Age = t_Age;
   }
   Hero Elder (Hero t_Hero)
   {
      if (Age > t_Hero.Age)
                return *this;
      else return t_Hero;
   }
   friend ostream & operator <<
            ( ostream& t_Out, Hero & t_Hero);
};
ostream & operator <<
            (ostream & t_Out,  Hero & t_Hero)
{
   t_Out << t_Hero.nm << " is ";
   t_Out << t_Hero.Age << " years old ";
   return t_Out;
}
```

only be overloaded as a friend as we want the object to be displayed to be passed as the second (and not the first) argument, we have done so. There are two elements of the Hero class, one describes the name and other the age. The main() function is equally straight forward. It just calls this member function Elder. The object returned is displayed next using cout.

```
Hero Elder (Hero t_Hero)
  {
     if (Age > t_Hero.Age)
            return *this;
     else return t_Hero;
  }
```

the rest of the code does not demand much explanation. We have overloaded << operator to display the information about the Hero. As this operator can

```
// Program 24.1 Block-1
int main()
{
   Hero Bajrangi("Salman",50);
   Hero Raj("Sharukh",49);

   Hero BigB = Bajrangi.Elder (Raj);
   cout << BigB << " and elder of the two \n";
}
```

[2] Modules 34,35 and 36 throws some light on the C++ object model, which describes many things including why such a transformation takes place.

Base and derived classes

Suppose we have the Shape, circle, rectangle and square classes defined as before. We can define a pointer to a Shape class and make it point to any other class down the hierarchy. We can also define a pointer to those derived classes. Let us try to understand the difference between those pointers.

```
Shape *pShape;
Circle *pCircle, SomeCIrcle;
```

Following are allowed

```
pShape = new Circle;
pCIrcle = new Circle;
```

Following is not allowed

```
pCircle = new Shape;
```

But we can cast the pointer before allocation and then it is allowed.

```
pCircle = (Circle *) pShape;
```

What exactly is the reason for this behavior of a C++ compiler? Allowing a base class pointer to point to derived class is allowed but not the other way round seems strange. Let us try to understand the reason. When we derive a class from another class, the derived class object contains the data members of both, base as well as a derived class in the fashion shown in 24.6. The data members of the derived class are kept immediately after the data members of the base class members. The part of the derived class object which contains the base class members is known as the base class subobject of the derived class. Whenever a pointer is defined, the pointer's access is confined to the type it is defined with. Both derived and base class members can point to the derived class object but while the derived class pointer can access every member of both, base class subobject as well as derived class object, the base class pointer can only access the base class subobject. Thus when `pCircle` access the circle object, it can access data members of Shape as well as data members of Circle, but `pShape` can only access Shape subobject.

This point needs further explanation. Every pointer, when defined, also defines the scope. The increment and decrement happen based on that definition. For example, if we have two pointers defined as int and char pointers.

```
int *pInt; char *pChar;
Emp *pEmp;
```

Here, ++pInt increments 64 bits(in a 64-bit compiler) to point to next int while ++pChar increments 8 bits to point to next char. If an Emp object is of 100 bytes, ++pEmp jumps 100 bytes to point to next Emp object.

Similarly, a derived class pointer, when incremented, increments according to the size of derived class while a base class pointer, when incremented, increments according to that size. That means, though both types of pointers can point to a derived class object, they are not same. Their scope and accordingly their process of increment, is different.

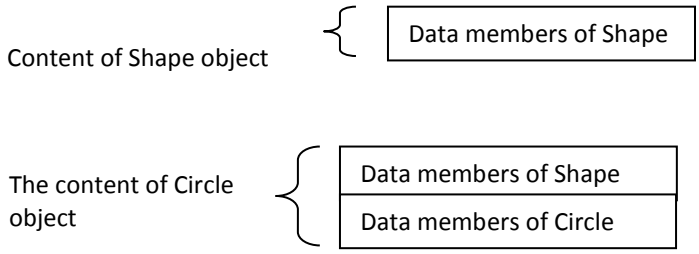

Figure 24.6 the inherited object layout

When Shape is inherited into Circle, it only contains one sub object of Shape class. Consider another case of multiple-inheritance. A cricketer class, as well as Indian class, inherits into an IndianCricketer class. If the object is defined as follows.

```
IndianCricketer Mithali;
```

The Mithali object contains two subobjects, Indian as well as Cricketer. In the case of a Square object, which is inherited from Rectangle, which in turn is inherited from Shape, it contains both subobjects. The compiler, when an object of one type is assigned to another, the compiler has to manage the process in such a way that the assignment makes sense. Figure 24.7 indicates how inherited and further inherited classes are laid out and the scope of the base class pointer in either case. In a case of multiple-inheritance, almost a similar structure sub object.

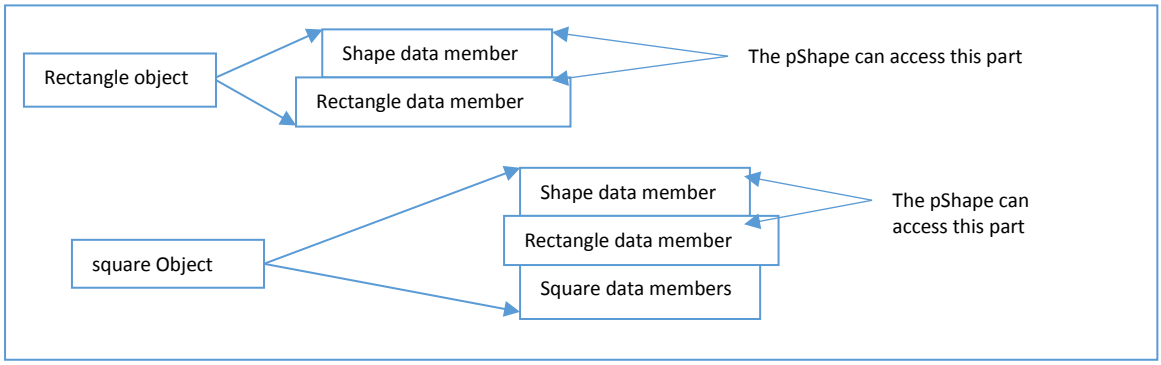

Figure 24.7 a further derived class and a scope of a pointer to base class

Derived class member function with the same name

What if a derived class member function has the same name that of the base class member function? The derived class object has now two members with the same name, one that it has inherited and another which it has defined itself. It can still access both of them using the scope resolution operator. What if we do not specify the scope resolution operator? The default function is the local function. Let us take an example to understand.

```
//Program 23.2 block 1
//NonVirtaul
#include <iostream>
using namespace std;
class Shape
{
   int l_Style;
   int fill;
public:

   void draw()
   {
      cout << "It is Shape \n";
   }
};
class Circle:public Shape
{
   int R;
   int CenterX;
   int CenterY;
public:

   void draw()
   {
      cout << "It is Circle \n";
   }
};
```

Closely observe the block 1 of program 24.2. There is two functions draw(), in Circle, first which is inherited from Shape and another that it has defined inside its body, both with the same name and thus are considered overloaded. However, our interest is to see what exactly is the output of the statements which invoke those functions.

We have three such statements in our main() described in the second block of the program 24.1. The output is shown at the end of that block indicating what is the output of some of the calls that we have made to draw() function associated with different objects or pointers.

First is,

```
t_Shape.draw();
```

It calls the draw() function of the Shape class. Another is following statement, which rightfully calls the Circle class draw() function.

```
Ring.draw();
```

When we have another set of statements, the behavior seems strange. pShape points to a circle object but calls a draw() function of the Shape class!

```
pShape = &Ring;

pShape->draw();
```

```
//Program 24.1 block 2
int main()
{
   Shape t_Shape, *pShape;
   Circle Ring;
   t_Shape.draw();
   Ring.draw();

   pShape = &Ring;

   pShape->draw();
}
```

It is Shape
It is Circle
It is Shape
Program ended with exit code: 0

When a base class pointer is pointing to a derived class object, it executes the BASE class function and not a derived class function.

A kind of local and global variable behavior is observed when we use an Object.Function notation. When two variables of the same name are used, in both, local and global context, the global variable is inferred when in a global context while in a local context, the local variable is in effect. However, when a pointer is pointing to an object and a function is invoked, there are two options for the compiler to act on.

1. The compiler decides the type of pointer by looking at the definition and choose the function belongs to that class.
2. The compiler, alternatively, can find out where the pointer is pointing to, and choose the function belongs to the object being pointed to at that point in time.

The C++ compiler, as you have seen in above case, chooses the first approach. The reason is, when a function is already defined and used to invoke the overloaded function like draw () above, the compiler has this pointer for this call available and can process the function call at compile time. When we define pShape is pointing to a Shape class, this pointer is this pShape itself (which is of type pointer to Shape). Unlike that, the second case requires the compiler to know where the pointer is pointing to which may not be possible at compile time. C++ compilers by default choose options which are runtime efficient and thus they choose the first version. What if we want to tell the compiler to go for the second option, that means, we want to the compiler to find out where the pointer is pointing to, and execute the function of that object? We can do so very easily by changing a single statement in the above program. The base class draw function definition now changes as follows.

```
class Shape
{..
   virtual void draw()
   …
It is Shape
It is Circle
It is Circle
..
```

Figure 24.8 Changes by virtual

The virtual word added just before the name of the function changes the default behavior. Now the compiler resolves the function based on the content the pointer is pointing to. The output now changes to as shown in 24.8. The third statement now indicates that it draws a circle. That means, the compiler now finds out where the pShape is pointing to and invokes the draw() function of that class.

Virtual functions

Virtual functions allow the process of choosing the function at runtime. Virtual functions are treated specially by compilers. When the virtual functions are deployed in the class, the class requires additional storage for two reasons. The first addition is called a virtual table. This table is used for storing pointers to all virtual functions. This virtual table is also known as vtble for short. Another additional storage requirement stems from the objects to need to have an access to this virtual table. Every object now has a pointer inserted as an additional member, known as vptr, pointing to that virtual table. Such a table is constructed for every class where at least a single virtual function is defined. All objects of that class invariably will have a single vptr as an additional member in all such cases. The idea is to reset this virtual

pointer to point to the vtble when the pointer points to object of another class. This resetting is done at runtime when pointers continue pointing to objects of different classes. When a different type of object is pointed to by the pointer, it has its vptr pointing to different vtble and thus can access to different virtual functions defined for those objects. Suppose following three statements are encountered.

```
Shape *pShape;
pShape = &Ring;
pShape->draw();
```

pShape is defined as pointer to shape in the first statement. When it is defined, it contains a vptr pointing to vtble of Shape class. When statement 2 is encountered and pShape is made to point to an object of class Circle, the pShape's vptr now is made to point to vtble of Circle class.

The virtual table is also created when the Run Time Type Information or RTTI is enabled, even when there is no virtual function defined. This is because this information is also stored in the same virtual table, usually as the first entry. Similarly, the information about virtual base classes is also kept either in this virtual table or an additional similar table is constructed. Point is, even when the virtual functions are not defined, there is a possibility of the virtual table to exist.

Constraints on virtual functions

Whenever designers want to have some functionality using virtual functions, he must learn about typical ways virtual functions can be defined and used.

1. The virtual functions, in the entire hierarchy, should bear the same prototype as the base class virtual function has, including the name. If a programmer makes a mistake of changing the prototype and keeping the same name, the function becomes a normal overloaded function and not a virtual function.
2. The derived class version of the virtual function, having the exact prototype as of the base class, may omit virtual keyword. It is considered by default.
3. A derived class can forgo defining a virtual function in their class, in that case, if ever the virtual function is called by either the object or the pointer to that object, it calls the base class virtual function. This is, as explained before, is like referring to the global variable when the local variable is not defined.
4. It is compulsory to define a virtual function in the base class. One can though define a virtual function with an empty body.
5. Defining a virtual function in a single class makes no sense. It is like defining a normal function. For any real life solution, the hierarchy must be present. The other requirement is technical; when a function is called, it must be called by the pointer to the object of the class or the reference of the class object and not the object. Only

when a virtual function is invoked by pointer or reference to the object, the polymorphism is possible to be achieved. Though, one can invoke a virtual function in a static fashion as well.

6. One cannot define a constructor as virtual. This is because when a derived class constructor is called, it calls the base class constructor as well to construct the base class subobject. Virtual functions of the derived class override the virtual functions of the base class, having just virtual constructor which replaces a base class constructor with a derived class won't do.
7. Virtual destructors are allowed and it is a good idea to have a virtual destructor. When a base class pointer points to a derived class object and executes a destructor, it is better it executes the derived class destructor and deletes the entire derived class object rather than the base class subobject.

The process of binding

The virtual functions are said to provide dynamic or late binding. Normal functions are resolved at compile time. That means, when a compiler encounters a function call, it can decide which is the exact function to be called, even when it is one of the overloaded functions, and convert it into that specific call. Thus the function call is bound to the exact function at compile time, known as static or early binding. Unlike that, when a virtual function is called, the compiler cannot decide which function is to be called at that point in time. The compiler converts that code to another code which runs and decides the function to execute at that point in time. Thus the function call is bound to the exact function at runtime.

The functions which are called from main () are looked at from the libraries used and linked with the main in the second phase after compilation. This process is known as linking, and the process which does so is called a linker. The trouble with the virtual function is that they are not possible to be resolved at compile time and thus linker cannot link them at compile time. That process is delayed till the runtime. That is why it is also called late binding.

Alternatives to virtual functions

There are a few possible alternatives to using virtual functions, sometimes. For example, one can use pointers to function or pointers to member functions. Using them, a lot of flexibility can be achieved without really needing the runtime overhead.

Another idea is to use a non-member non-virtual function which is independent of the class to access the situation at runtime and decide what to do, instead of using a virtual function. For example, if we have some treasures and health values associated with the video game that we were discussing, it is intuitive for us to call a virtual function as follows.

```
pPlayer->getPoints()
```

The player hierarchy decides the points based on who the player is and few other attributes.

In fact, we do not need to define such a virtual function. We may define another non-member non-virtual function. getPoints(Player) to look at what is the public information available about the player and return points the player has earned. Similarly, one can also define similar non-member non-virtual function initPoints(Player) for initiating points, addPoints(Player) for adding points and subPoints(Player) for subtracting points for a given player.

The above-mentioned process may not be feasible if the required information is not publicly available or the situation demands real runtime decision making but the point is, one must look for alternatives to virtual functions before making a decision.

Summary

In this module, we have looked at the difference between a compile time and runtime polymorphism. We have seen that this pointer is available to a function to have an access to the invoking object. We have seen how the hierarchy of the classes can redefine functions of the base class with the same name. When a function is redefined with a virtual keyword attached in the beginning, we can call it a virtual function. A virtual function, unlike a normal function, is resolved at runtime. We have seen an example of how one can define and use virtual functions in a program in this module. We finished with a note that a programmer should be able to look for alternatives to the virtual function as they offer flexibility but at the cost of performance at runtime.