Chapter 9

Polymorphism, Pointers & Virtual Function

Polymorphism

Polymorphism is one of the crucial features of OOP's. It simply means one name, multiple forms. The concept of polymorphism is implemented using overloaded function and operators. The overloaded member functions are selected for execution by matching arguments with their type and no.

This information is known to the compiler at the compile time. Therefore compiler is able to select the appropriate function. This is called Early Binding or Static Binding or Static Linking. It is also known as compile time polymorphism.

Now let us consider a situation where the function name and prototype is the same in the both the base and derived classes.

e.g.
```
        class A
        {
                int x;
            public :
                void show( )
                {
                    . . . .
                }
        };

        class B
        {
                int y;
            public :
                void show( )
                {
                    . . . .
                }
        };
```

Since the prototype of show( ) is the same in both the places, the function is not overloaded and therefore static binding does not apply. In fact, the complier does not know what to do and defers the decision.

It would be nice if the appropriate member function could be selected while the program is running. This is known as runtime polymorphism. C++ supports a mechanism known as Virtual Function to achieve runtime polymorphism. At run time, when it is known what class objects are under consideration, the appropriate version of the function is called.

Pointers to Objects

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 runtime. Dynamic binding is one of the powerful features of C++. This requires the use of pointers to object.

The pointer variables can be used to point to locations of built-in data type. Such as int, float etc. It is possible for us to create pointer which points locations of user define data type.

e.g.

```
class item
{
        int no;
        float cost;
    public :
        void getdata( );
        void putdata( );
};
```

The statement item *p; will create a pointer variable p which will store the address of item type location. i.e. p will point to an object of class item. So p is known as a pointer to an object.

Consider the following statement.

```
item x,*p;
p = &x  ;
p->getdata( );
p->putdata( );
```

Here the pointer variable p contains the address of an object x. The member functions of class item can be referred by using the pointer with the arrow operator. i.e. The statement p->getdata( ) & p->putdata( ) will call the member functions for the object pointer by p.

Write a Program to Demonstration of Pointer to Object.

```
#include<iostream.h>
#include<conio.h>

class item
{
  private:
      int code;
      float price;
  public:
      void getdata()
      {
       cin>>code>>price;
      }
```

```
       void show()
       {
        cout<<"Code = "<<code<<endl;
        cout<<"Price = "<<price<<endl;
       }
};

main()
{
clrscr();
item *p=new item[2];
item *d=p;
for(int i=1;i<=2;i++)
{
   cout<<"Enter code & price for item"<<i<<" ";
   p->getdata();
   p++;
}
for(i=1;i<=2;i++)
{
   cout<<"Item = "<<i<<endl;
   d->show();
   d++;
}
getch();
return(0);
}
```

<u>this</u> <u>Pointer</u>

C++ uses a unique keyword called 'this' to represent an object that involves a member function. 'this' is a pointer that points to the object which calls a member function.

e.g.

```
        a1.getdata( );
```

This statement will set the pointer 'this' to the address of object a1.

This unique pointer is automatically pass to a member function when it is called the value of 'this' pointer acts as an implicit arguments to all the member functions. In the member function we refer each data member simply by specifying its name. But computer actually consider the data member along with 'this' pointer.

e.g.

```
        class ABC
        {
                int a;
                int b;
            public :
                void getdata( );
        };
```

```
void ABC : : getdata( )
{
    cin>>a>>b;
}
```

Instead of ‘cin>>a’ computer assumes the statement as ‘cin>>this->a’ & instead of ‘cin>>b’ it assumes ‘cin>>this->b’.

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. The statement

```
return *this;
```

inside a member function will return the object that invoked the function. This statement assumes important when we want to compare two or more objects inside a member function and return the invoking object as a result.

Write a Program to Demonstration of ‘this’ Pointer.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>

class person
{
  private:
      char name[20];
      int age;
  public:
      person(char *s,float a)
      {
       strcpy(name,s);
       age=a;
      }
      person & person :: greater(person &x)
      {
       if(x.age>=age)
          return x;
       else
          return *this;
      }
      void display()
      {
       cout<<"Name = "<<name<<endl
           <<"Age = "<<age;
      }
};
```

```
main()
{
clrscr();
person p1("Amol",20),p2("Amit",24),p('\0',0);
p=p1.greater(p2);
cout<<"Elder person is"<<endl;
p.display();
getch();
return(0);
}
```

Virtual Function

Polymorphism refers to the property by which object belonging to different classes are able to response to the same message but in different form. A single pointer variable can be used to refer to the object of different classes i.e. A base class pointer can be used to store the address of a derived class object.

But when address of a derived class object is stored in the pointer to the base class object, it always executes the function in the base class. The compiler simply ignores the contents of the pointer & select the member function that matches the type of the pointer. In such a situation the concept of virtual function is used.

When we use the same function name in base & derived class, the function in the base class is declared as virtual. A function is declared as virtual just by preceding a keyword virtual to the function name. When a function is made virtual, C++ determines which function should be called at runtime. Here the pointer variable checks the type of object pointed by the base class pointer.

Write a Program to Demonstration of Virtual Function.

```
#include<iostream.h>
#include<conio.h>

class base
{
  public:
      void show()
      {
       cout<<"Show Base"<<endl;
      }
      virtual void display()
      {
       cout<<"Display Base"<<endl;
      }
};

class der : public base
{
  public:
      void show()
```

```
        {
        cout<<"Show Derived"<<endl;
        }
        void display()
        {
        cout<<"Display Derived";
        }
};

main()
{
clrscr();
base b,*bp;
der d;
cout<<"bp points to Base"<<endl;
bp=&b;
bp->show();
bp->display();
cout<<"bp points to Derived"<<endl;
bp=&d;
bp->show();
bp->display();
getch();
return(0);
}
```

Rules for Virtual Function

When virtual functions are created for implementing late binding, we should observe some basic rules.

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 prototype of the base class version of a virtual function and all the derived class version must be identical. If two functions with the same name have different prototype, 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 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.

Array of Pointers to Objects

We can also create an array of objects using pointers. For example, the statement

item *ptr = new item[10];

creates memory space for an array of 10 object of item. Remember, in such cases, if the class contain constructors, it must also contain an empty constructor.

Write a Program to Demonstration of Array of Pointer to Object.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>

class city
{
  private:
       char *name;
       int len;
  public:
       city()
       {
        len=0;
        name=new char[len+1];
       }
       void getname()
       {
        char *s;
        cout<<"Enter City name = ";
        cin>>s;
        len=strlen(s);
        name=new char[len+1];
        strcpy(name,s);
       }
       void display()
       {
        cout<<name<<endl;
       }
};

main()
{
city *cp[10];
clrscr();
int n=1;
int op;
do
{
   cp[n]=new city;
   cp[n]->getname();
   n++;
   cout<<"Do you want to Continue ?(1/0)";
   cin>>op;
}while(op);
cout<<"City names"<<endl;
for(int i=1;i<=n;i++)
   cp[i]->display();
getch();
return(0);
}
```

Pure Virtual Function

It is normal practice to declared 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, 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 complier 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. Such class are called Abstract base Classes.

Pointers to Derived Classes

We can use pointers not only to the base objects but also to the objects of derived classes. Pointers to objects of a base class are type-compatible with pointers to 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 *cptr;        // pointer to class B type variable
B b;            // base object
D d;            // derived object
cptr = &b;      // cptr points to object b
```

We can make cptr to point to the object d as follows.

cptr = &d;

This is perfectly valid with C++ because d is an object derived from the class B.

However, there is a problem in using cptr to access the public members of the derived class D. Using, cptr, we can access only those members inherited from B and not the members that originally belong to D. In case a member of D has the same name as one of the members of B, then any reference to that member by cptr will always access the base class member.

Although C++ permits a base pointer to points to any object derived from that base, the pointer cannot be directly used to access all the members of the derived class. We may have to use another pointer declared as pointer to the derived type.

Write a Program to Demonstration of Pointer to Derived Objects.

```
#include<iostream.h>
#include<conio.h>

class base
{
  public:
      int b;
      void show()
      {
       cout<<"B = "<<b<<endl;
      }
};

class der : public base
{
  public:
      int d;
      void show()
      {
       cout<<"B = "<<b<<endl
           <<"D = "<<d<<endl;
      }
};

main()
{
clrscr();
base *bp;
base ba;
bp=&ba;
bp->b=100;
cout<<"bp points to Base Object"<<endl;
bp->show();
der de;
bp=&de;
bp->b=200;
cout<<"bp points to Derived Object"<<endl;
bp->show();
der *dp;
dp=&de;
dp->d=300;
cout<<"dp is Derived type Pointer"<<endl;
dp->show();
cout<<"Using ((der *)bp)"<<endl;
((der *)bp)->d=400;
((der *)bp)->show();
getch();
return(0);
}
```