Lesson : 10 Virtual Functions And Templates

Objectives

After completing the lesson you will understand about:

- Types of polymorphism
- Defining the pointer to objects and functions
- Overriding the base class functions by virtual functions
- Generic programming using function templates and class templates

Structure Of The Lesson

10.1 Introduction To Polymorphism And Virtual functions

Polymorphism means “one name with multiple forms”. The process of resolving or mapping a function call with function definition is known as “binding”. In general, the function calls are resolved or mapped to function definitions when the program is complied. This is called “compile time binding” or “compile time polymorphism” . When virtual functions are called on objects they are not resolved at compile time, and postponed till the execution of program. Hence it is called “late binding” or “dynamic binding”.

Dynamic binding is one of the powerful features of C++. This process uses the pointer to objects. We will see how pointers to objects and virtual functions are used to implement dynamic binding.

10.2 Pointers To Objects

A pointer can point to an object created by a class.
e.g:

```
class item
{
    int code;
    float price ;
          public:
          void getdata(int,float);
          void show();
 }
item i1;
item * ptr;
ptr=&i1;
 ptr= new item;
```

“**item** “ is a class with int code and float price. **getdata()** and **show(**) are its member functions. **i1** is an object of type item. ***ptr** is a pointer pointing to the type item. The item i1 is accessed by **i1.getdata()** . It can also be accessed using the pointer variable pointing to that function i.e., **ptr->getdata()**. When the statement,

```
                    ptr = new item;
```

is executed it allocates enough memory for the data members in the object structure and assigns the address of member space to the pointer ptr.

// A sample program to access the object through pointers

```
#include<iostream.h>
           class item
            {
                     int code;
                     float  price ;
                      public  :
                            void  getdata (int  a, float b )
                             {
                                     code =a;
                                      price = b;
                              }
                             void show ( )
                             {
                                    cout<<"code:"<< code <<endl;
                                     cout<<"price:"<<price<<endl;
                              }
          };
void main ( )
{
        item  *p= new item[2];
        item   *d = p;
         int x, i ;
        float y;
         for ( i =0 ;    i <2 ; i++ )
            {
                  cout<<"Enter code & item \n:";
                   cin>>x>>y;
                   p-> getdata ( x, y );
                    p++;
             }
            p=d;
            for ( i= 0 ; i<2 ; i++)
               {
                      cout << " item:\n";
```

```
                        p->show ( );
                        p++;                    }
                d= NULL;
                delete [ ] p;

}
```

output:

```
        Enter code & item
        :1
        2.2
        Enter code & item
        :3
        3.4
         item:
        code:1
        price:2.2
         item:
        code:3
        price:3.4
```

10.3 Pointers To The Derived Class

Pointers can be used to point to the base class as well as the derived class objects. A single pointer can point to different classes.

For e.g.:

```
Let B be the base class and D is the derived class.
B * bptr ; // pointer to the base class
B       b ;  // Base class object
D       d ; // Derived class  object
bptr = &b; // pointer points to the  object of the base class B
bptr =&d; // pointer points to the object of the derived class D.
```

By pointing the base class pointer to the derived, the member inherited by the base class can be accessed but the members of derived class cannot be accessed.

Program to demonstrate the baseclass pointer pointing to the derived class

```
# include <iostream.h>
class base
{
  public : int b;

   void show( )
    {
        cout<<"This is a base class \n";
           cout<<"b= "<<b<<"\n";
     }

};

class derived : public base
  {
       public : int d;

        void show( )
         {
            cout<< " This is derived ";
            cout << "b = " <<b;
            cout<< "d = "<<d;
         }
};

void main ( )
{
    base * bptr;
    base bs;
    bptr = & bs;
    bptr -> b = 100;
    bptr -> show ( );

    derived  der ;
    bptr = &der;
     bptr-> b =50;
     //bptr->d = 300       //doesnot work
    bptr->show();

    derived  * dptr;
          dptr = &der;
     dptr -> d = 70;
     dptr -> show ( );  }
```

output:

```
This is a base class
b= 100
This is a base class
b= 50
 This is derived class b = 50d = 70
```

10.4 Overriding The Base Class Functions Using Virtual Function

When a derived class redefines a base class member function the member function call on the objects uses the definition given in derived class. But whether the same member function is called on the derived object using a base class type pointer it leads to the function definition in the base class to be executed instead of the definition given in derived class. The compiler ignores the contents of the pointer and chooses the member function that matches the pointer. To avoid this, the base class version of the function should be declared as a “virtual” function. This is done by, inserting the keyword “**virtual**” to the function declaration in base class. The usage of virtual functions leads to late binding of function calls i.e., function calls to the virtual functions are resolved at the time of running the program. This is also called “execution time binding” or “late binding” or “runtime polymorphism”.

When the function, the function executed at the runtime is based on the type of the object pointed to, by the base pointer rather than the type of the pointer.

Note: Coercion is the automatic conversion of one type of value to another type.

Program to demonstrate virtual functions

```
#include<iostream.h>

class bc
{
public:
 void  display()
 {
```

```
 cout<<"Display base\n";
 }
  void virtual show()
 {
 cout<<"This is base\n";
  }
};
class dc :public bc
{
public:
 void display()
 {
 cout<<"Display derived\n";
 }
 void show()
 {
 cout<<"This is derived \n";
 }
};

void main()
{
bc *bptr;
bc bas;
cout<<”Baseclass pointer is pointing to baseclass
object\n”;
bptr = &bas;
bptr->display();
bptr->show();
dc *dptr;
dc der;
cout<<”Baseclass pointer is pointing to derived class
object\n”;
bptr = &der;
bptr->display();
bptr->show();
cout<<”Derived class pointer is pointing to derived class
object\n”;
dptr = &der;
dptr->display();
dptr->show();
}
```

output:

```
Baseclass pointer is pointing to baseclass object
Display base            //Base class functions are executed
This is base
Baseclass pointer is pointing to derived class object
Display base            //Baseclass function is executed
This is derived//As it is virtual, derived class function is executed
Derived class pointer is pointing to derived class object
Display derived         //Derived class functions are executed
This is derived
```

Program To Demonstrate Hierarchical Inheritance Using Virtual Functions:

```
//Gshape class provides basic functionality for any geometric shape. It
//contains a virtual function area, which is overridden in each of the
//derived classes.
        # include < iostream.h>
        #include<conio.h>
        class Gshape
        {
              public:
                 virtual double area ( ) {return 0;}
        };
        class Triangle : public Gshape
        {
              double base, height;
              public:
                  Triangle ( ) {base = height>0.0;}
                   Triangle (double  b, double h)
                   {
                        base = b; height =h;
                    }
                    double area ( )
                    {
                    return (0.5 * base * height ) ;
                     }
        };
        class Circle : public Gshape
        {
                  double radius;
              public :
                  Circle ( ) {radius =0.0 ;}
                   Circle (double r)
```

```
                {
                    radius =r;
                 }
                 double area ( ) ;
        };
        class Rectangle : public Gshape
        {
                 double length, breadth;
                  public:
                         Rectangle ( ) {length = breadth =0.0;}
                         Rectangle (double l, double b)
                          {
                               length =l, breadth =b;
                           }
                         double area ( );
        };
        double Circle ::area ( )
        {
                 return (3.141 * radius * radius );
         }
        double Rectangle :: area ( )
        {
                 return (length * breadth );
        }
        void main ( )
        {

                 Triangle t(5.5, 3.5 );
                 Circle     c(7.0);
                  Rectangle   r (8.0, 9.5);
             Gshape * objects [3];
                   objects[0] = &t;
                   objects[1] =&c;
                   objects[2] = &r;
                                              clrscr();
          cout<<"area of triangle, circle,rectangle are_ _ _";
                      for (int i=0; i<3; i++ )
                            cout << "\t"  << objects [i] ->area ( );
        }
```

output:

```
        area of triangle, circle,rectangle are_ _ _ 9.625   153.909 76
```

10.5 Templates

Templates are an important feature of C++. It provides greater flexibility to the language by supporting generic programming. This allows the reusable of software components such as functions, classes, etc., supporting different data type in a single framework.
A template is like a blueprint, which can be used to create a number of similar objects. It allows the construction of family of template functions and classes to perform the same operation on different data type. The templates declared for functions are called function templates and those declared for class are called class templates.

10.5.1 Function Templates

A function template in C++ is a sub program defined using a generic algorithm and generic data types. A generic algorithm is an algorithm that can be applied to different types of data to achieve similar results. In C++ we use function templates to generate similar functions for different data types. A generic function defines the general set of operation that will be applied to various types of data. It has the type of data that will operate upon when passed as a parameter. The same general procedure can be applied to a wide range of data. The generic function is independent of any data. The compiler automatically generates the correct code for the type of data that is actually used when the function is executed.

The definition of the template begins with " template <class T>" it is called template prefix. It tells the compiler that the definition or prototype that follows this statement is a template. "T" is a type parameter. The word class means the type or data type. The type parameter 'T' can be replaced by any type i.e., it may be simple data type or user defined data type. Within the body of the function definition the type parameter "T" is used just like any other data type.

The Syntax For Defining Function Templates:

```
 template <class    type_ parameter>
 Return type function name (parameter ;list);
```

The parameter list should contain at least one parameter of generic type.

e.g.:

```
            Template<class  T>
           Return type       void  swap(T&V1, T&V2);
```

The definition of a template function is written as any other function definition. The function header will begin with the template prefix. The template function can be called from the main program like any function call.

Consider swapping of two variables. The swapping algorithm is same for any type of variables. In a generic type a swap function can be defined as follows:

e.g.:

```
              template<class T>
              void swap(T&a,T&b)
               {
              T    t;
               t = a;
               a= b;
               b= t;
               }
```

The above function template can be applied to swap values of two integer variables by replacing “T” with “int”. The same templates can also be applied to swap the values of two objects of a given type (class) by substituting the class name for “T”(variable type). Like this the above function template can be used to swap values of any type of variables.

Function template is used with different types of data, as the complier will produce a separate definition for each different type that we use with the function template. The function call statement for template function is written like any other function.

For Example:

```
   swap (ch1, ch2);           // ch1 & ch2 are character variables
   _  _   _  _
   _ _  _  _  _
   swap(f1, f2);              // f1&f2  are floating point variables
```

Program To Swap The Contents Of Two Variables Using A Template Function

```
#include<iostream.h>
template<class T>
void swap(T& var1, T& var2)
{ T temp;
temp = var1;
var1 =var2;
var2 =temp;
}
int main()
{
int int1 = 1, int2 =2;
cout<<"Integer values are "<<int1<<" and "<<int2<<endl;
swap(int1,int2);
cout<<"Integer values after swapping are "<<int1<<" and "
<<int2<<endl;
double d1 = 1.1, d2 =2.2;
cout<<"Double  values are "<<d1<<" and "<<d2<<endl;
swap(d1,d2);
cout<<"Double values after swapping are "<<d1<<" and" << d2
<<endl;
char symb1 = 'A', symb2 ='B';
cout<<"Character values are "<<symb1<<" and "<<symb2<<endl;
swap(symb1,symb2);
cout<<"Character values after swapping are "<<symb1<<" and
"<<symb2<<endl;
return 0;
}
```

output:

```
Integer values are 1 and 2
Integer values after swapping are 2 and 1
Double  values are 1.1 and 2.2
Double values after swapping are 2.2 and 1.1
Character values are A and B
Character values after swapping are B and A
```

Program For Swapping A List Of Values With Bubble Sort Technique Using Template Functions

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

template <class T >
```

```
void swap(T&v1,T&v2)
{
          T    t;
         t=v1;
         v1=v2;
         v2=t;
}
//generic function to display a list of values.
template<class T>
void showlist(T list[],int size)
{
      int i=0;
      for(;i<size;i++)
      cout<<'\t'<<list[i];
}

//template function to sort list of values
template<class T>
void bsort(T list[],int size)
{
       int i,j;
        for(i=0;i<size-1;i++)
          for(j=0;j<size-1;j++)
              if(list[j]>list[j+1])
        swap(list[j],list[j+1]);
}
void main()
{
     float list1[]={10.f,-11.5f,13.0f,2.1f,8.4f};
             int list2[]={3,11,5,2,1};
     cout<<"\n\t floats before sort"<<endl;
     showlist(list1,5);
     bsort(list1,5);
     cout<<"\n\t floats after sort"<<endl;
     showlist(list1,5);
     cout<<"\n\tintegers before sort"<<endl;
     showlist(list2,5);
             bsort(list2,5);
     cout<<"\n\tintegers after sort"<<endl;
     showlist(list2,5);
}//main()
```

output:

```
    floats before sort
```

```
10      -11.5   13      2.1     8.4
 floats after sort
-11.5   2.1     8.4     10      13
integers before sort
3       11      5       2       1
integers after sort
1       2       3       5       11
```

Template Function For Binary Search:

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

//template function for show_list()
template<class T>
void showlist(T list[],int size)
{
              for(int i=0;i<size;i++)
                     cout<<"\t"<<list[i];
}
//template function for binary search
template<class T>
void bisearch(T list[], T se,int low,int high,int& pos)
{
          int mid;
           if(low<=high)
           {
                  mid=(low+high)/2;
                    if(list[mid]==se)
                                                 pos=mid;
                    else if(list[mid]< se)
                    {
                                low=mid+1;
                                    bisearch(list,se,low,high,pos);
                     }
                     else            if(list[mid]>se)
                     {
                                 high=mid-1;
                                 bisearch(list,se,low,high,pos);
                     }

            }
}
```

```
void main()
{
            int pos=-1;
             float    list1[]={-11.5f,10.2f,13.5f,22.6f};
                                  int list2[] = {4,12,26,31};
               cout<<"\n\t float type data";
                showlist(list1,4);
                bisearch(list1,22.6f,0,3,pos);
                cout<<"\n\t element found at:"<<pos+1;
                                             cout<<"\n\t integer data";
                showlist(list2,4);
                bisearch(list2,26,0,3,pos);
             cout<<"\n\t element found at:"<<pos+1;
}
```

output:

```
        float type data        -11.5   10.2    13.5    22.6
        element found at:4
        integer data   4       12      26      31
        element found at:3
```

10.5.2 Class Templates

Templates are also used in defining generic classes like generic algorithm. We can define generic classes and later substitute in the generic class to get more specific classes. A generic class is defined with type parameters. The syntax of class template starts with a template header and continues with the class definition.

Syntax for class templates:

```
template <class type parameters>
class class_name
{
            generic variables;
                 .
                                  .
                 .
                 .
           generic functions;
}
```

The generic class consists of generic member variables and functions. The class and member function definitions are same as for any ordinary classes expect the type parameter has to be used in place of data type. The member functions and overloading operators are defined as function templates. In the main program a type argument is given to the class template to make the object specific.

Program to demonstrate a class template for representing a pair of values.

```
#include<iostream.h>
template<class T>
class pair
{
        private:
          T   first,second;
        public:
        pair() { }
        pair (T a, T b)
         {
                first =a;
                 second=b;

          }
      void set (int pos, T  v);
        friend   ostream & operator<<(ostream & out,pair<T> p);
};
   //template  function for overloading stream operator.
   template<class T>
  ostream & operator<<(ostream&out,pair<T> p)
{
      out<<p.first<<"   "<<p.second;
       return out;
}
template < class T>
void pair<T>::set(int pos,T   v)
{
            if(pos==1)
                      first =v;
             else if(pos==2)
                       second=v;
}
void main()
{
          pair<int>      intpair;
```

```
                    pair<char>     charpair('A','B');
                    cout<<"\n\t The pair of characters:"<<charpair;
                     intpair.set(1,10);
                     intpair.set(2,20);
                     cout<<"\n\t The pair of integers:"<<intpair;
          }
```

output:

```
        float type data       -11.5   10.2    13.5    22.6
        element found at:4
        integer data   4       12      26      31
        element found at:3
        The pair of characters:A   B
        The pair of integers:10   20
```

Class templates are used for instantiation by substituting a data type name for a type parameter. This substitution creates a specific definition of the template class and leads to the creation of objects of that specific type, for example,”pair<int> int pair “ creates int pair object to contain a pair of integers. This is because the data type int is substituted for the type parameter name ‘T’. We can use “typedef” keyword to define types based on a specific data type substitution in the template class. For example

```
         typedef pair<int> int pair;
```

defines ‘int pair’ as a class of pair of integers by substituting int for the type parameter ‘T’. This data type can be used to instantiate objects for integer pairs i.e.,

```
         int pair  p1;
```

declares p1 as an object that contains pair of integers.

A class template program to process a list of values

```
          #include<iostream.h>
          template<class T>
          class list
          {
                      public:
                              list ()
                                 {     itemcount=0;}
                              void  add(T ele);
                                   int isfull ();
                               //return 1 if it is full otherwise 0;
                              int length();
```

```
                friend ostream& operator<<(ostream& out,list<T> l);
        private:
                        T contents[20];
                         int itemcount;
};
template<class T>
 void list <T>::add(T ele)
{
                if(!isfull())
                {
                            contents[itemcount]=ele;
                             itemcount++;
                 }
                else
                          cout<<"\n  the list is full.";
}
template<class T>
ostream& operator<<(ostream& out,list<T> l)
{
               int i;
                for(i=0;i<l.itemcount;i++)
                          out<<l.contents[i]<<'\t';
                return out;
}
 template <class T>
int list<T>::isfull()
{
            if(itemcount==20)          return 1;
             else                             return 0;
 }
template<class T>
 int list<T>::length()
{
             return itemcount;
}

void main()
{
              list<int>li;
              list<char>lc;
              li.add(10);
              li.add(20);
              li.add(30);
              cout<<endl<<"The list of integers:"<<li;
              lc.add('A');
```

```
                    lc.add('B');
                     lc.add('C');
                     lc.add('D');
                     cout<<endl<<"The list of charactcers:"<<lc;
                     cout<<"\nThe total no of elements in both the
lists:"<<li.length()+lc.length();
          }
```

output:

```
          The list of integers:10 20  30
          The list of charactcers:A      B       C       D
          The total no of elements in both the lists:7
```

A program to demonstrate Inheritance Of Class Templates: To Prepare A Set Class And To Manipulate It.

```
#include<iostream.h>
#include<conio.h>
const int MAXSIZE=20;
template<class T>
class bag
{
          protected:
                    T   contents [MAXSIZE];
                    int itemcount;
          public:
                    bag()
                                {
                                itemcount=0;
                                }
                    int isempty()
                    {
                       if(itemcount==0)
                                          return   1;
                                 else
                                          return 0;
                     }
                     int  isfull()
                     {
                       if(itemcount==MAXSIZE)
                                          return 1;
                                else
                                          return 0;
                       }
                    void put(T item)
                     {
                       if(!isfull())
                          contents[itemcount++]=item;
                     }
                     int isexist(T ele);
                     void show();
};

template<class T>
 int bag<T>::isexist(T  item)
{
               int i;
               for(i=0;i<itemcount;i++)
                     if(contents[i]==item)
                        return 1;
                return  0;
}
template<class T>
void bag<T>::show()
{
             int i=0;
             for( ; i<itemcount;i++)
               cout<<" "<<contents[i];
               cout<<endl;
}
template<class T>
class set:    public bag<T>
    {
            public :
                     void add(T item);
                     void read();
                     void operator=(set <T> s);
                     friend set <T> operator +(set  <T> s1, set<T> s2);
     };
template<class T>
 void set<T>::add(T item)
           {
           if(!isexist(item) && !isfull())
                     put(item);
}
template<class T>
 void set<T>::read()
{
           T item;
```

```
             char   ch;
               while(1)
                {
                      cout<<"\n\t input an element:";
                      cin>>item; add(item);
                      cout<<"\t\t another (Y/N) ?";
                      cin>>ch;
                      if((ch=='N' )|| (ch=='n'))
                         break;
                 } //end of while statement.
} //end of function.

template<class T>
void set<T>::operator=(set<T> s)
{
               int i;
               itemcount= s.itemcount;
               for(i=0;i<itemcount;i++)
                  contents[i]=s.contents[i];
}
template<class T>
set<T> operator +(set <T> s1, set<T> s2)
{
            set <T> temp;
            temp=s1;
             for(int i=0;i<s2.itemcount;i++)
                     temp.add(s2.contents[i]);

                          return   temp;
}

void main()
{
               set <int>  s1;
                                       set <int> s2;
                                       set <int> s3;
               set <char> s4;
                cout<<"\n\t  enter  integers for set1";
                 s1.read();
                cout<<"\n\t enter integers for set2";
                  s2.read();
                  s3=s1+s2;
                 cout<<endl<<"s1=";
                  s1.show();
                 cout<<endl<<"s2=";
```

```
                  s2.show();
                cout<<endl<<"s1 union s2:";
                  s3.show();
                cout<<"\n\t enter input characters of set u:";
                  s4.read();
                cout<<endl<<"s4=";
                  s4.show();
}
```

output:

```
          enter  integers for set1
         input an element:1
                another (Y/N) ?y

         input an element:2
                another (Y/N) ?n

         enter integers for set2
         input an element:2
                another (Y/N) ?y

         input an element:3
                another (Y/N) ?n

s1= 1 2

s2= 2 3

s1 union s2: 1 2 3

         enter input characters of set u:
         input an element:e
                 another (Y/N) ?y

         input an element:t
                another (Y/N) ?n

s4= e t
```

10.6 Summary

- Polymorphism means one name having multiple forms.
- Types of polymorphism namely compile time and run time polymorphism are studied.
- Object pointers are useful in creating objects at run time.
- Run time polymorphism is achieved using virtual functions.
- Generic programming using templates by defining function templates or class templates

10.7 Technical Terms

Class template: It is a generic class definition.

Dynamic Binding: The selection (binding) of the function is done dynamically at the runtime. The compiler is able to select the appropriate overloaded member function made in the function call.

Function template: It is a generic function definition.

Instantiation: A process of creating a specific class from a class template.

Template: A feature of C++ that allows generic programming.

Template class: A class create from a class template .

Template function: A specific function created from a function template.

Virtual function: A function qualified by the 'virtual' keyword. When a virtual function is called, the class of the object pointed to determines which function definition has to be used.

10.7 Model Questions:

1. Write notes on virtual functions.
2. What is a Function template? Explain the syntax of function template.

3. What is class template? Explain the syntax of function template.
4. What is algorithm abstraction.

10.8 References

Object-oriented programming with C++,
by E. Bala Gurusamy.
Problem solving with C++
by Walter Savitch
Mastering C++
by K.R.Venugopal, RajkumarBuyya, T.RaviShankar

AUTHOR

M. NIRUPAMA BHAT, MCA., M.Phil.,
Lecturer,
Dept. Of Computer Science,
JKC College,
GUNTUR.