OBJECT ORIENTED PROGRAMMING

(with C++)

Unit-3

Generic Function

Unit-III

Inheritance, Class hierarchy, derivation – public, private & protected; aggregation, composition vs classification hierarchies

polymorphism, categorization of polymorphic techniques, method polymorphism, polymorphism by parameter, operator overloading, parametric polymorphism,

generic function – template function, function name overloading, overriding inheritance methods, run time polymorphism.

Templates

- Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.
- Templates allows function and classes to operate with generic types.
- *There are 2 type of templates:*
  - *Function templates* – A function template is a *"generic"* function that can work with different data types. L*ike macros, It enables software reuse. But unlike macros, fn templates help eliminate many type of errors through scrutiny of full c++ type checking.*
  - *Class templates* - Allows type-specific versions of generic classes. Just like function templates, we can also define class templates.

Templates

- ***Template Parameter*** - A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameter which is used to pass values to a function, template parameter allows to pass values and also types to a function.
- ***Template Instantiation*** – *When the compiler generates a class, function or static data members from a template, it is referred to as template instantiation.*
- *A function generated from a function template is called a generated function.*

Function Templates

Function generated from function template is called template function.

Syntax:

```
template <class T> returnType FnName(Parameter List)
   //Template function
{
   :
}
```

Templates function with 2 arguments of same/different type depending on argument passed e.g.

```
template <class T, class U> T getMin(T a, U b)
{
   return (a>b?a:b);
}
```

Function Templates

Example:

```
#include <iostream.h>
template<class x> void swap(x &a, x& b)
{
      x temp;
      temp=a;
      a=b;
      b=temp;
}
```

```
main()
{
      int i=10,j=20;
      double x=10.1,y=21.3;
      char a='x', b='z';
      swap(i,j);
      swap(x,y);
      swap(a,b);
      cout<<i<<j;
      cout<<a<<b;
      cout<<x<<y;
}
```

Function Templates

- A function template can be overloaded as well as overrided.
- Example of function Template overloading:

```
#include<iostream.h>
template <class t> void Max(t a, t b)
{
      if(a>b)
          cout<<a;
      else
          cout<<b;
}
```

```
template <class t> void Max(t a, t b, t c)
{
        if(a>b && a>c)
            cout<<a;
        else if(b>a && b>c)
            cout<<b;
        else
            cout<<c;
}
main()
{
    Max(1,2);
    Max(3,2,1);
}
```

Function Templates

- If a template is invoked with a user defined type, and if that template uses functions or operators (e.g. ==,+,<= etc.) with objects of that class type, then those functions and operators must be overloaded for the user-defined type. Forgetting to overload such operators causes compilation error.

Example

Program to define the function template for calculating the square of given numbers with different data types.

```
template <class T> T square(T number)
{ return number * number; }
int main()
{
// Get an integer and compute its square
int iValue;
cout << "Enter an integer value: ";
cin >> iValue;
// compiler creates int square(int) at call to square with an int argument
cout << "The square is " << square(iValue);  //square<int>(iValue);
// Get a double and compute its square
cout << "\nEnter a double value: ";
double dValue;
cin >> dValue;
// compiler creates double square(double)on call to square with double arg
cout << "The square is " << square(dValue);  // //square<double>(dValue);
return 0;
}
```

Class Templates

- Class created from a class template is called template class
- Creating a class from class template is called Instantiation
- Syntax:

```
template <class t> class className
{
      :
}
```

For objects:

```
className <Type> objectName(Arguments);
```

e.g.

```
 className <int> obj1(10);
className <float> obj2;
```

Class Templates

- Non-Type Parameters & Default Type

Non-Type Parameter

```
template <class T, int Size>
    class Array
{
      T a[size];
      :
};
```

For Object:

```
      Array<int,10> obj1;
            //Instantiation
      Array<float, 5> obj2;
```

Default type

```
template <class T= int> class Array
{
      T a[5];
      :
};
```

For Object:

```
      Array<> obj;  //Instantiation
```

Class Templates

- Member function templates - Syntax:

```
template <class T> retType className<T>::fnName(Arguments)
{
    :
}
```

e.g.

```
template<class T>
T Point<T> :: operator *(Point & P)
{
   :
}
```

Example

Program to find the bigger of two entered numbers using class template.

```
template <class T> class pair
{
        T a; T b;
        public:
        pair()
        {
        cin>>a>>b;
        }
        T get_max();
};
Template<class T>
T pair<T>::get_max()
{
        T ret;
        ret=a>b?a:b;
        return ret;
}
```

```
void main()
{
        cout<<"\nEnter 2 Integer numbers:";
        pair<int> obj1;
        cout<<"Greatest Integer is”
        <<obj1.get_max();
        Cout<<“Enter 2 Float Numbers:\n";
        pair<float> obj2;
        cout<<"Greatest Float is”
        <<obj2.get_max();
}
```

Templates & Inheritance

A class template can be derived from a template/non-template class

Example

```
template <class T> class Base
{
    protected:
            T var;
    public:
            Base(T val): var(val)      { }
            T Get()  { return var;      }
};
template <class T> class Derived : public Base<T>
{
    public:
            Derived(T valu): Base<T>(valu)  { }
            T fun()   {  return var;      }
};
```

Function Templates & Static Variable

Each instantiation of function template has its own copy of local static variables.

Example

```
template <class T> void fun(const T &x)
{
    static int i=10;
    cout<< ++i;
}
main()
{
    fun<int> (1);   //prints 11
    fun<int> (2);   //prints 12
    fun<double> (1.1);      //prints 11
}
```

Output:

11
12
11

Class Templates & Static Variable

Each instantiation of class template has its own copy of member static variables.

Example

```
template <class T> class Test
{
    private:
            T val;
    public:
    static int count;

    Test()
    {
            count++;
    }
};
Template<class T> int Test<T>::count=0;
```

```
main()
{
    Test<int> a;      //Value of count for
                        Test<int> is 1 now
    Test<int> b;      //Value of count for
                        Test<int> is 2 now
    Test<double> c;    //Value of count for
                        Test<double> is 1 now
    cout<<Test<int>::count;  //prints 2

    cout<<Test<double>::count;//prints 1
}
```

End of Unit-3