Object Oriented Programming in C++

Nepal College of Information Technology

Course Instructor: Er. Rabina Chaudhary

Chapter 6: Template and generic programming

Generic Programming:

- A new concept that enables us to define generic classes and functions and thus provides support for generic programming
- Generic programming is an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures

Template:

- C++ allows defining general classes or functions to handle different data types
- The generic family of classes and functions is called template

- In template, we define function with general type of arguments such that we can use the function for any data type
- A template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the function

Example:

a template for a function multiply() multiplies int, float, double types

Template:

- In template, we define class with general type data member such that we can use the class for any data type
- When an object of a specific type is defined for actual use the template definition for that class is substituted with the required data type

Example:

- a class template for an array class enables us to create array of various data types such as int or a float

Types of Template:

1. Function template
2. Class Template

Function Template:

- Function template is used to define a function to handle arguments of different types
- Function templates are special functions that can operate with generic types
- In C++ this can be achieved using template parameters
- A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

Without using function template:

```
#include<iostream.h>
#include<conio.h>
void swap(int a, int b)
{
        int temp;
        temp=a;
        a=b;
        b=temp;
        cout<<"After  swaping a="<<a<<endl<<"   b="<<b<<endl<<endl;
}

void swap(float a,float b)
{
        float temp;
        temp=a;
        a=b;
        b=temp;
        cout<<"After  swaping a="<<a<<endl<<"   b="<<b<<endl<<endl;
}
```

```
void swap(char a, char b)
{
            char  temp;
            temp=a;
            a=b;
            b=temp;
            cout<<"After swaping a="<<a<<endl<<"    b="<<b<<endl<<endl;
}

void main()
{
            int r=100,s=200;
            swap(r,s);
            float x=100.2323,y=121.333;
            swap(x,y);
            char a='a',b='b';
            swap(a,b);
getch();
}
```

Defining function template:

1. Defining general data type:

```
template < typename templateName>
```

2. Defining a function with general datatype

```
returnType functionName (argument_with_template_name)
{
    body of function with type template_name if appropriate
}
```

Example Program:

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

template <typename T> //defining template
void swap(T a, T b) //using generic type T wherever appropriate in function template
{
        T temp;
        temp=a;
        a=b;
        b=temp;
        cout<<"After swaping a="<<a<<endl<<"   b="<<b<<endl<<endl;
}
void main()
{
        int r=100,s=200;
        swap(r,s);
         swap<float>(102.560,45.1212);
        char a='a',b='b';
        swap(a,b);
getch();
}
```

Practice:

1. Write a program to find maximum among 3 integers, 3 double values and 3 characters using function template
2. Write a program to sort integer array, character array and float array using function template

Function template with multiple parameters:

- More than one generic data type in function template
- They are separated by comma separated list

1. Defining general data type

```
template < typename template_name1, typename template_name1,…>
```

2. Defining function with general data type

```
returntype functionName (arguments with template_name1, arguments with template_name2, …)
{
        body of function with type template
}
```

Example Program:

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

template <typename T1,typename T2>
void testFunction( T1 x, T2 y)
{
		cout<<endl<<x<<endl<<y<<endl<<endl;
}

void main()
{
		testFunction(123,124.656);
		 testFunction(10.234,'z');
		testFunction(12,'r');
		getch();
}
```

Practical:

Q. Define a function that calculates the memory allocated for integer, double, float, char using function template with multiple parameters.

Overloading function templates:

- A template function may be overloaded either by template function or ordinary function of its name
- In normal function overloading, the function may have the same definition, but with different arguments types or numbers or sequence
- In case of templates, the overloading happens only on the basis of number of parameters

Example program: Function template overloading

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

template <typename T>
T maximum( T x, T y)
{
            T z= (x>y)?x:y;
            return z;
}
template <typename T>
T maximum ( T x, T y , T z)
{
            if( x>y && x>z)
                        return x;
            else if(y>z)
                        return y;
            else
                        return z;
}
```

```
template <typename T>
void display(T x)
{
        cout<<endl<<x<<"is maximum"<<endl;
}


void main()
{
        int max=maximum(123,1246,1234);
        display(max);
        char Cmax=maximum('a','b');
        display(Cmax);
 getch();
}
```

Template function can also be overloaded by non- template function:

- Template function can be overloaded with ordinary or non-template function
- In this case, the compiler gives priority to ordinary function with exact match, if the exact match is not with the ordinary function, then compiler prefers the template function

Template function can also be overloaded by non- template function: Example

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

int  maximum(  int  x, int y)
{
                    cout<<endl<<"normal  function is called"<<endl ;
                    int z= (x>y)?x:y;
                    return  z;
}
template  <typename  T>
T maximum(  T x, T y)
{
                    cout<<endl<<"template  function with two parameters is called"<<endl;
                    T z= (x>y)?x:y;
 return  z;
}
```

```
template <typename T>
T maximum ( T x, T y , T z)
{
        cout<<endl<<"template function  with three parameters is called"<<endl;
        if( x>y && x>z)
                return x;
        else if(y>z)
                return y;
        else
                return z;
}
template <typename T1>
void display(T1 x)
{
        cout<<endl<<"template function is called to display maximum"<<endl;
        cout<<endl<<x<<" is maximum"<<endl;
}
```

```
void main()
{
        int max=maximum(123,1246,1234);
        display(max);
        cout<<endl<<endl<<"For maximum among two integers"<<endl;
        max=maximum(10,20); /*calling non-template function (ordinary
                                function)*/
        display(max);
        cout<<endl<<endl;
        char Cmax=maximum('a','b');
        display(Cmax);
  getch();
}
```

Practical:

Q1. Write a program to overload template function to swap two values and three values. The values to be swapped must be integer, char and double. Also implement template overloading with ordinary function.

Q2. Write a function template to calculate average and multiplication of numbers

Q3. Calculate roots of quadratic equation by using function template

Non type template argument:

- A template non-type parameter is a special type of parameter that does not substitute for a type, but is instead replaced by a value

Example:

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

template  <typename  T, int size>  //size is non-type  parameter

void testArray(T x[])
{
		int i;
		cout<<endl<<"Enter  values of array elements  "<<endl;
		for(i=0;i<size;i++)
		{
		cin>>x[i];
		}
		cout<<endl<<"The  array is :";
		for(i=0;i<size;i++)
		{
			cout<<x[i]<<",";
		}
}
```

```
void main()
{
        int x[5];
        double y[10];
        char z[10];
        cout<<endl<<"For integer array"<<endl;
        testArray<int, 5>(x);   //generic type T is replaced by int
                                  //non-type parameter, size, is replaced by 5
         cout<<endl<<endl<<"For double array"<<endl;
        testArray<double, 10>(y);         //generic type T is repalced by double
                                          //non-type size is replaced by 10
        cout<<endl<<endl<<"For character array"<<endl;
        testArray<char, 5>(z);
getch();
}
```

Class Template:

- A class template is a kind of class which has members of template type
- Class template lets us define the behavior of a class without actually knowing what data type will be handled by the operations of the class
- Template class provide the mechanism for creating applications with generic type which are common applications such as linked list, stacks ,queues, etc.

General form for class template:

Step 1. defining template

```
template<typename T>
```

Step 2: Defining class with members of template type

```
class className
{
        class members with data type  T whenever appropriate
}
```

Step 3: Defining object

```
class_name <specific data type for T> objectName;
```

Class Template:

- Each object of template class must be of some specific size. Therefore, the data type must be specified before creating an object of template class

Example Program:

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

template  <typename  T>
class Rectangle
{
            T length, breadth;
            public:
                        Rectangle();
                        Rectangle (T x, T y)
                        {
                        length=x;
                        breadth=y;
                        }
                        T area();
};
```

```
template  <typename  T>
Rectangle<T>::Rectangle()
    {
                length=0;
                breadth=0;
    }
template  <typename  T>
  T Rectangle<T> :: area()
    {
                return  (length*breadth);
    }

void main()
{
                Rectangle <int> obj1(5,89);
                Rectangle <double>  obj2(12.34,56.78);
                cout<<obj1.area()<<endl;
                cout<<obj2.area()<<endl;
                cout<<sizeof(obj1)<<endl<<sizeof(obj2);
  getch();
}
```

Class Template with multiple parameters:

- We can define class template with more than one generic data type
- They are declared as comma separated list in template specification

Step 1. defining template

```
template<typename T1, typename T2,…>
```

Step 2: Defining class with members of template type

```
class className
{
        class members with data type  T1, T2, … whenever appropriate
}
```

Step 3: Defining object

```
class_name <specific data type for T1, specific data type for T2, …> objectName;
```

Example Program:

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

template <typename T1, typename T2>
class Rectangle
{
  T1 length;
  T2 breadth;
  public:
                    Rectangle();
    Rectangle (T1 x, T2 y)
    {
                    length=x;
      breadth=y;
    }
    T1 area();
};
```

```
template <typename T1,typename T2>
Rectangle<T1,T2>::Rectangle()
    {
                    length=0;
                    breadth=0;
    }
template <typename T1,typename T2>
  T1 Rectangle<T1,T2> :: area()
    {
                    return (length*breadth);
    }

void main()
{
  Rectangle <int,double> obj1(5,89.12);
  Rectangle <double,double> obj2(12.34,56.78);
  Rectangle <int,int> obj3(1,2);
  cout<<obj1.area()<<endl;
  cout<<obj2.area()<<endl;
  cout<<obj3.area()<<endl;
  cout<<sizeof(obj1)<<endl<<sizeof(obj2);
  getch();
}
```

Non-type template argument:

```
template <class T, int size>
 class className
            {   T a[size];
                  …
            };
```

This template supplies the size of the array as an argument

So we can make objects as

```
 className <int, 10>   a1;  // array of 10 integers
 className <float, 10> a2; // array of 10 floats
className <char, 20>  a3;  // string of size 20
```

Member function of class template defined outside class:

Syntax:

```
template<typename T>
returntype  className<T> :: functionName(argument_list)
{
        body of function
}
```

Practical:

Q1. Define a class Array with general type data member and appropriate member functions to input, display and sort the array. Use this class as to perform operation on integer array, character array and double array.

Q2. Define a class Box with l, b and h as general type data members and appropriate functions to calculate volume of a box. Use template class for the above operations.

Practical:

3. Define two classes “Polar” and “Rectangle” to represent points in polar and rectangle system. Use conversion routines to convert from one system to another system using template.

Implementing stack data structure :

```
Step 1 : start
Step 2 : Include all the header files which are used in the program
Step 3 :Define a class to represent Stack with following class members
        Step 3.1 : Create a one dimensional array with fixed size (int stack[SIZE])
        Step 3.2 : Declare all the functions used in stack implementation
        Step 3.3: Define a integer variable 'top' and initialize with '-1'. (int top = -1) in constructor
Step 4: In main function, make the function call to insert element to stack and remove element from stack as
per the requirement
Step 5: stop
```

push(value) - Insert value into the stack

- In a stack, push() is a function used to insert an element into the stack.
- New element is always inserted at top position.
- Push function takes one integer value as parameter and inserts that value into the stack

Step 1 - Check whether stack is FULL. (top == SIZE-1)

Step 2 - If it is FULL, then display "Stack is full and insertion is not possible”

Step 3 - If it is NOT FULL, then increment top value by one (top++) and set stack[top] to value (stack[top] = value).

pop() : remove element from stack

- In a stack, the element is always deleted from top position.

Step 1 - Check whether stack is empty. (top == -1)

Step 2 - If stack is empty, then display "Stack is EMPTY"

Step 3 - If it is not empty, then remove stack[top] and
decrement top value by one (top--).

Program implementing stack operation using class template:

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
template <typename T, int max>
class Stack
{
                         T stk[max];
                        int top;
                        public:
                        Stack()
                        {
                                                top = -1;
                        }

                        void push(T data);

                        void pop();

                        void show();
};
```

```
template<typename  T, int max>
void Stack<T,max> ::push(T data)
 {
            if (top == (max -1))
                        cout << "stack is full\n";
            else
            {
            top++;
            stk[top]= data;
            }
 }
template<typename  T, int max>
void Stack<T,max>:: pop()
 {
            if (top == -1)
                        cout << "stack is empty\n";
            else
            {
                        T data = stk[top];
                        cout<<endl<<data  << " is popped  out"<<endl;
                        top--;
            }
}
```

```
template<typename T, int max>
void Stack<T,max>:: show()
        {
                for(int i=top;i>=0;i--)
                cout << "\nstack["<< i <<"]" << stk[i];
        }
```

```
void main()
{
                    Stack <char ,5>s;
                   int choice;
                   char data;
 while(1)
 {
                   cout<<endl<<endl<<"***Menu***"<<endl <<"1. push data "<<endl<<"2. Pop data"<<endl<<"3. Display"<<endl<<"4. exit"<<endl;
                   cout<<"Enter your choice : ";
                   cin>>choice;
                   switch(choice)
                   {
                   case 1:          cout<<"Enter data to be pushed into stack :  " ;
                                   cin>>data;
                                   s.push(data);
                                   break;
                   case 2:         s.pop();
                                   break;
                   case 3:          s.show();
                                   break;
                   case 4: exit(0);
                   default : cout<<"Please enter valid choice"<<endl;
                   }
}

 }
```

Implementation of stack using standard template library:

```
#include<iostream>
#include<stack>

using namespace std;

int main()
{
        stack<int> stackobj;
        stackobj.push(41);
        stackobj.push(42);
        stackobj.push(45);
        stackobj.push(50);
        while (!stackobj.empty()) {
        cout << endl << stackobj.top()<<endl;
        stackobj.pop();
 }
```

```
        cout<<endl;
        cout<<endl;

        stackobj.push(541);

        stackobj.push(50);
        stackobj.pop();
        while (!stackobj.empty()) {
        cout << " " << stackobj.top();
        stackobj.pop();
  }
 return 0;
}
```

Implementing Stack using standard template library:

```
#include<iostream>
#include<stack>
using namespace std;
int main(){
        stack<int> stackobj;
        stackobj.push(41);
        stackobj.push(42);
        stackobj.push(45);
        stackobj.push(50);
        while (!stackobj.empty()) {
        cout << endl << stackobj.top()<<endl;
        stackobj.pop();
  }
        cout<<endl;
        cout<<endl;
        stackobj.push(541);

        stackobj.push(50);
        stackobj.pop();
        while (!stackobj.empty()) {
                cout << " " << stackobj.top();
                stackobj.pop();
        }
    return 0;
}
```

Standard Template Library:

Standard template library is a set of C++ template classes to provide template classes and functions that implement many popular and commonly used algorithms and data structures like vectors, stack, queues, lists , etc.

- Components of STL
  - Containers
  - Algorithms
  - Iterators

Containers:

- Containers are used to manage collections of objects of a certain kind
- There are several different types of containers like queue, stack, list, etc

Algorithms:

- Algorithms acts on containers
- They provides means by which we will perform operations on contents of containers
- Algorithm library contains built in functions that performs complex algorithms on the data structures

Iterators:

- Iterators are used to point to the containers
- Acts as bridge between containers and algorithm