UNIT 5

Templates

Templates

- **Template** 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.
- A *t*emplate can be used to create a family of classes or functions .For example, a class template for an array class enables us to create array of various data types such as **int** array or a float array. We can define template for a function, say mul(),that would help us create various versions of mul() for multiplying int, float and double type values.
- A template *can be* considered *as* a kind of a macro .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. since, a template is defined with a ***parameter*** that would be replaced by a specified data type at the time of actual use of the class or function. So a templates are also called **a *parameterized classes* or *functions.***

There are two types of templates:
1. Function Templates
2. Class Templates

CLASS TEMPLATE

We can create class templates for generic class operations. Sometimes, we need a class implementation that is same for all classes, only the data types used are different. Normally, we would need to create a different class for each data type or create different member variables and functions within a single class. This promotes the coding redundancy and will be hard to maintain, as a change is one class/function should be performed on all classes/functions. However, class templates make it easy to reuse the same code for all data types.
The general format of class template is

```
template <class T>
class class_name
{
//class member  specification
//with anonymous type T
//whenever appropriate
}
```

The prefix template <class T> tells the compiler that we are going to declare a template and use T as the type name in the declaration. T may be substituted by any data type including the user defined types.

While creating object of class, it is necessary to mention which type of data is used in that object.

Syntax for creating object

The syntax for creating an object of a template class is

class_name<datatype>   object_name;

Example:

- *For integer data* ***sample<int>   s1****; Here s1 is object of class sample*
- *For float data* ***sample<float> s2****; Here s2 is object of class sample*
- *For char data* ***sample<char>   s3****; Here s3 is object of class sample*

WAP to add two integer and two floating point numbers using class templates.

```
#include<iostream>
using namespace std;
template<class T>
class sample
{
private:
T a,b,s;
public:
sample(T x,T y)
{
a=x;
b=y;
}
void calculate()
{
s=a+b;
}
void display()
{
 cout<<"sum="<<s<<endl;
}
};
int main()
{
 sample<int> s1(5,8);
 sample<float> s2(3.5,8.9);
 cout<<"For integer values:"<<endl;
 s1.calculate();
 s1.display();
 cout<<"For float values"<<endl;
 s2.calculate();
 s2.display();
 return 0;
}
```

WAP to perform sum and product of two integer and two floating point number using class template.

```
#include<iostream>
using namespace std;
template<class T>

class sample
{
private:
T a,b,s,p;
public:
sample(T x,T y)
{
a=x;
b=y;
}
void calculate()
{
s=a+b;
p=a*b;
}
void display()
{
cout<<"sum="<<s<<endl;
cout<<"product="<<p<<endl;
}
};

int main()
{
 sample<int> s1(5,8);
 sample<float> s2(3.5,8.9);
 cout<<"for integer values:"<<endl;
 s1.calculate();
 s1.display();
 cout<<"for float values:"<<endl;
 s2.calculate();
 s2.display();
 return 0;
}
```

WAP to find the maximum value between two integers and two floating point numbers using class template.

```
#include<iostream>
using namespace std;
template<class T>
class compare
{
	private:
	T a,b;
	public:
	compare(T x,T y)
	{
	a=x;
	b=y;
	}
	void max()
	{
		if(a>b)
		{
		 cout<<"maximum value="<<a<<endl;
		}
		else
		{
		  cout<<"maximum value="<<b<<endl;
		}
	}
};

int main()
{
 compare<int> c1(5,6);
 compare<float> c2(4.9,32.4);
 cout<<"For integer values"<<endl;
 c1.max();
 cout<<"For float values"<<endl;
 c2.max();
 return 0;
}
```

Create a class template to find the scalar product of vectors of integers and vectors of floating point number.[PU:2009 spring]

```
#include<iostream>
using namespace std;
template <class T>
class vector
{
private:
T a,b,c;
public:
vector(T x,T y,T z)
{
	a=x;
	b=y;
	c=z;
}
T operator *(vector p)
{
	T sum;
	a=a*p.a;
	b=b*p.b;
	c=c*p.c;
	sum=a+b+c;
	return sum;
}
void display()
{
	cout<<a<<"i+"<<b<<"j+"<<c<<"k"<<endl;
}
};
int main()
{
vector<int>  v1(5,6,7),v2(9,10,11);
cout<<"v1=";
v1.display();
cout<<"v2=";
v2.display();
cout<<"scalar product of  integer values="<<v1*v2;
vector<float>  m(1.1,2.2,3.3),n(5.5,6.6,7.7);
cout<<"m=";
m.display();
cout<<"n=";
n.display();
cout<<"scalar product of  float values="<<m*n;
return 0;
}
```

Class template with multiple parameters

We can use more than one generic data type in a class template. They are declared as a comma-separated list within the template specification as shown below.

```
template<class T1,class T2>

class class_name
{
. . . . . . . . . . . . . . . .
 (Body of the class)
. . . . .  . .  .  . . . . . . .
};
```

Example:

```
#include <iostream>
using namespace std;
template<class T1,class T2>
class Test
{
T1 a;
T2 b;
public:
Test(T1 x, T2 y)
{
a=x;
b=y;
}

void show()
{
cout<<a<<" "<<b<<endl;
}
};
int main()
{
Test<float,int> test1(1.23,123);
Test<int,char> test2(100,'W');
test1.show();
test2.show();
return 0;
}
```

WAP to perform sum of two integer numbers, two floating point numbers , one integer and one floating point numbers using class template.

```
#include<iostream>
using namespace std;
template<class T1, class T2>
class sample
{
private:
T1 a;
T2 b;
public:
sample(T1 x,T2 y)
{
a=x;
b=y;
}
void add()
{
cout<<"sum="<<(a+b)<<endl;
}
};

int main()
{
 sample<int,int> s1(5,8);
 sample<float,float> s2(3.5,8.9);
 sample<int,float>  s3(3,8.9);
 cout<<"sum of two integer data="<<endl;
 s1.add();
 cout<<"sum of two float type data="<<endl;
 s2.add();
 cout<<"sum of integer and float data="<<endl;
 s3.add();
 return 0;
}
```

Define a class called stack and implement generic methods to push and pop elements from stack.[PU:2015 fall]

```
#include <iostream>
using namespace std;
#define MAX_SIZE 10
template <class T>
class stack
{
	private:
	T stk[MAX_SIZE];
	int top;

	public:
	stack()
	{
	top = -1;
	}
	void push(T data)
	{
		if (top == (MAX_SIZE -1))
		{
		cout << "stack is full"<<endl;
		}
		else
		{
		top++;
		stk[top]= data;
		}
	}

	void pop()
	{
	if (top == -1)
	{
		cout << "stack is empty"<<endl;
		}
		else
		{
		top--;
		}
	}
```

```
        void show()
        {
                for(int i=top;i>=0;i--)
                {
                cout << "stk["<< i <<"]="<<stk[i]<<endl;
                }
        }
};

int main()
{
stack<char> s;
s.push('a');
s.push('b');
s.push('c');
s.push('d');
s.push('e');
s.push('f');
s.show();
cout << "popped top"<<endl;
s.pop();
s.show();
return 0;
}
```

Function template

- Function template can be used to create a family of functions with different argument types.
- A single function template can work with different data types.
- Any type of function argument is accepted by the function.

The general format of function template is:

```
template <class T>
return_type    function_name(arguments of type T)
{
……………………
//body of function with type T
//wherever appropriate
……………………
}
```

1. Program to display maximum value among two integer and two floating point numbers using function templates.

```
#include <iostream>
using namespace std;
template <class T>
void compare(T x,T y)
{
	if(x>y)
	{
		cout<<"maximum value="<<x<<endl;
	}
	else
	{
	   cout<<"maximum value="<<y<<endl;
	}
}

int main()
{
	int i1=4, i2=5;
	float f1=50.6, f2=10.5;
	cout <<"for integer numbers"<<endl;
	compare(i1,i2);
	cout <<"for floating point numbers"<<endl;
	compare(f1,f2);
	return 0;
}
```

2. Create a function template to swap two values.[PU:2018 fall]

```
#include <iostream>
using namespace std;
template <class T>
void swapvar(T &x,T &y)
{
	T temp;
	temp=x;
	x=y;
	y=temp;
}
```

```
int main()
{
        int i1=10,i2=20;
        float f1=5.5,f2=8.2;
        cout <<"Before swapping"<<endl;
        cout <<"i1="<<i1<<" "<<"i2="<<i2<<endl;
        cout <<"f1="<<f1<<" "<<"f2="<<f2<<endl;
        swapvar(i1,i2);
        swapvar(f1,f2);
        cout<<"After swapping"<<endl;
        cout<<"i1="<<i1<<" "<<"i2="<<i2<<endl;
        cout<<"f1="<<f1<<" "<<"f2="<<f2<<endl;
        return 0;
}
```

3. Write a function templates to calculate the average and multiplication of numbers.[PU 2012 spring]

```
#include<iostream>
using namespace std;
template<class T>
void calculate(T a,T b)
{
 T pro;
 float avg;
 avg=(a+b)/2.0;
 pro=a*b;
 cout<<"Average="<<avg<<endl;
 cout<<"Product="<<pro<<endl;
}
int main()
{
        int i1=10,i2=25;
        float f1=9.4,f2=4.3;
        cout<<"Calculation for integer numbers"<<endl;
        calculate(i1,i2);
        cout<<"Calculation for float numbers"<<endl;
        calculate(f1,f2);
        return 0;
}
```

4. Write a function template to calculate the sum and average of numbers.[PU:2009 fall]

```
#include<iostream>
using namespace std;
template<class T>
void calculate(T a,T b)
{
 T sum;
 float avg;
 sum=a+b;
 avg=(a+b)/2.0;
 cout<<"Average="<<avg<<endl;
 cout<<"Sum="<<sum<<endl;
}
int main()
{
	int i1=10,i2=25;
	float f1=9.4,f2=4.3;
	cout<<"Calculation for integer values"<<endl;
	calculate(i1,i2);
	cout<<"Calculation for float values"<<endl;
	calculate(f1,f2);
	return 0;
}
```

5. Create a templates to find the sum of two integers and floats.

[PU:2014 fall] [PU:2016 spring] [PU:2017 spring]

```
#include<iostream>
using namespace std;
template <class T>
void sum(T x,T y)
{
T s;
s=x+y;
cout<<"sum="<<s<<endl;
}
```

```
int main()
{
int i1,i2;
float f1,f2;
cout<<"Enter two integer number"<<endl;
cin>>i1>>i2;
sum(i1,i2);
cout<<"Enter two float number"<<endl;
cin>>f1>>f2;
sum(f1,f2);
return 0;
}
```

6. WAP to find the roots of quadratic equation using function template.

```
#include<iostream>
#include<math.h>
using namespace std;
template<class T>
 void calculate(T a,T b,T c)
 {
        T d=b*b-4*a*c;

         if(d<0)
         {
         cout<<"Roots are imaginary"<<endl;
         }
         else if(d==0)
        {
                cout<<"Roots are  real and equal"<<endl;
                cout<<"R1=R2="<<(-b/(2.0*a));
         }
         else
         {
         cout<<"Roots are real and unequal"<<endl;
                float r1=(-b+sqrt(d))/(2.0*a);
                float r2=(-b-sqrt(d))/(2.0*a);
                cout<<"R1="<<r1<<endl;
                cout<<"R2="<<r2<<endl;
         }
 }
```

```
int main()
{
    int a1,b1,c1;
    float a2,b2,c2;
    cout<<"Enter the integer coefficient "<<endl;
    cin>>a1>>b1>>c1;
    calculate(a1,b1,c1);
    cout<<"Enter the float coefficient "<<endl;
    cin>>a2>>b2>>c2;
    calculate(a2,b2,c2);
    return 0;
}
```

7.Write a program to find the sum of integer and float array using function templates.

```
#include<iostream>
using namespace std;
template<class T>
T sum(T a[],int size)
{
    T s=0;
    for(int i=0;i<size;i++)
    {
        s=s+a[i];
    }
    return s;
}
int main()
{
    int x[5]={10,20,30,40,50};
    float y[3]={1.1,2.2,3.3};
    cout<<"Integer array element sum="<<sum(x,5)<<endl;
    cout<<"Float array element sum="<<sum(y,3)<<endl;
    return 0;
}
```

Note:

A function generated from a function template is called template function. Program Demonstrates the use of template functions in nested form for implementing bubble sort algorithm.

Bubble sort using Template functions (Using of template functions using nested form)

```
#include<iostream>
using namespace std;
template<class T>
void bubble(T arr[],int n)
{
for(int i=0;i<n-1;i++)
{
         for(int j=0;j<n-i-1;j++)
         {
          if(arr[j]>arr[j+1])
          {
          swap(arr[j],arr[j+1]);
           }
         }
}
}
template <class X>
void swap(X &a,X &b)
{
 X temp;
 temp=a;
 a=b;
 b=temp;
}
int main()
{
 int i,j;
 int x[5]={50,10,30,20,40};
 float y[5]={1.1,5.5,3.3,4.4,2.2};
  bubble(x,5);
  bubble(y,5);
  cout<<"Sorted x-array"<<endl;
  for(i=0;i<5;i++)
   {
     cout<<x[i]<<endl;
   }
   cout<<"Sorted y-array"<<endl;
   for(i=0;i<5;i++)
    {
      cout<<y[i]<<endl;
    }
     return 0;
}
```

Write a program to find the maximum elements in an array using function template.

```
#include<iostream>
using namespace std;
template<class T>
T find_max(T arr[],int n)
{
        T max=arr[0];
        for(int i=0;i<n;i++)
        {
         if(arr[i]>max)
         {
         max=arr[i];
          }
        }
        return max;
 }

int main()
{
int imax;
float fmax;;
int x[6]={50,10,30,20,40,70};
float y[5]={1.1,5.5,3.3,4.4,2.2};
imax=find_max(x,6);
fmax=find_max(y,5);
cout<<"Maximum value in integer array="<<imax<<endl;
cout<<"Maximum value in float array="<<fmax<<endl;
return 0;
}
```

Function templates with multiple parameters

We can use more than one generic data type in template statement using a comma separated list as shown below.

```
Template<classs T1,class T2,…….>
return_type function_name(arguments of types T1,T2. . . .  . . )
{
. . . . . . . . . . . . .
(Body of function)
. . . . . . . . . . . . . . .
}
```

Program to illustrate the concept of two generic data types.

```
#include<iostream>
using namespace std;
template <class T1,class T2>
void display(T1 x,T2 y)
{
 cout<<x<<" "<<y<<endl;
}
int main()
{
 cout<<"calling function template with integer and string type parameters"<<endl;
 display(199,"pokhara");
 cout<<"calling function template with float and integer type parameters"<<endl;
 display(12.34,5);
 return 0;
}
```

Write a program to add two integers, two floats and one integer and one float numbers respectively. [PU:2005 fall]

```
#include<iostream>
using namespace std;
template <class T1,class T2>
void sum(T1 x,T2 y)
{
cout<<"sum="<<(x+y)<<endl;
}
int main()
{
int i1,i2,i3;
float f1,f2,f3;
float s1,s2,s3;
cout<<"Enter two integer values"<<endl;
cin>>i1>>i2;
sum(i1,i2);
cout<<"Enter two  float values"<<endl;
cin>>f1>>f2;
sum(f1,f2);
cout<<"Enter one  integer and one float values"<<endl;
cin>>i3>>f3;
sum(i3,f3);
return 0;
}
```

Merit and demerit of using template in C++

Merit

- C++ templates enable us to define a family of functions or classes that can operate on different types of information.
- We can use templates in situations that result in duplication of the same code for multiple types. For example, we can use function templates to create a set of functions that apply the same algorithm to different data types.
- Deliver fast, efficient, and robust code
- Easy to use
- When we use templates in combination with STL – it can drastically reduce development time.

Demerit

- Historically, some compilers exhibited poor support for templates. So, the use of templates could decrease code portability.
- Automatically generated source code can become overwhelmingly huge.
- Compile-time processing of templates can be extremely time consuming.
- It can be difficult to debug code that is developed using templates. Since the compiler replaces the templates, it becomes difficult for the debugger to locate the code at runtime.
- Templates are in the headers, which require a complete rebuild of all project pieces when changes are made.
- Many compilers lack clear instructions when they detect a template definition error. This can increase the effort of developing templates, and has prompted the development of Concepts for possible inclusion in a future C++ standard.
- No information hiding. All code is exposed in the header file. No one library can solely contain the code .
- Though STL itself is a collection of template classes, templates are not used to write conventional libraries.

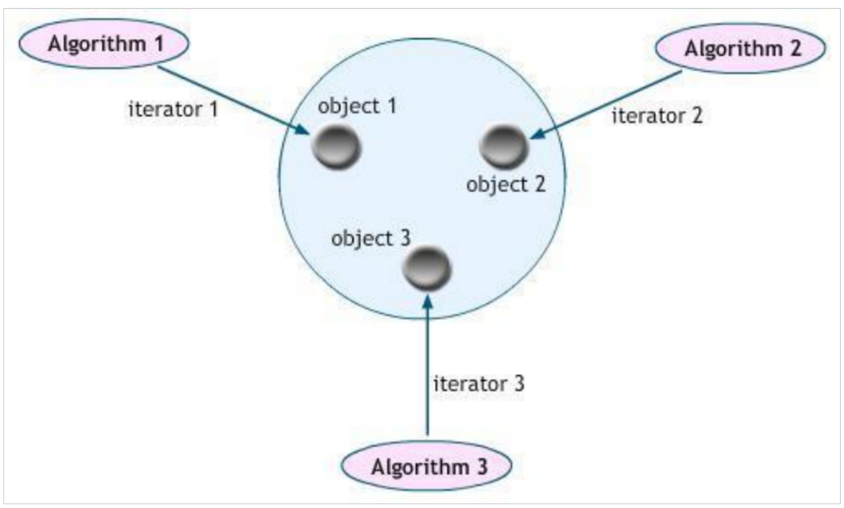
Standard Template Library(STL)

- In order to help the C++ users in generic programming, Alexander Stepanov and Meng Lee of Hewlett-Packard developed a set of general purpose templatized class (data structure) and functions (algorithms) that could be used as a standard approach of storing and processing of data. The collection of these generic classes and functions is called the Standard Template Library (STL).
- It helps to save C++ users' time and effort there by helping to produce high quality programs.

Components of STL:

The STL contains several components. But at its core are three key components. They are
1. Containers
2. Algorithms
3. Iterators

These three components work in conjunction with one another to provide support to a variety of programming solutions. The relationship between the three components is as shown in figure below. Algorithms employ iterators to perform operation stored in containers.



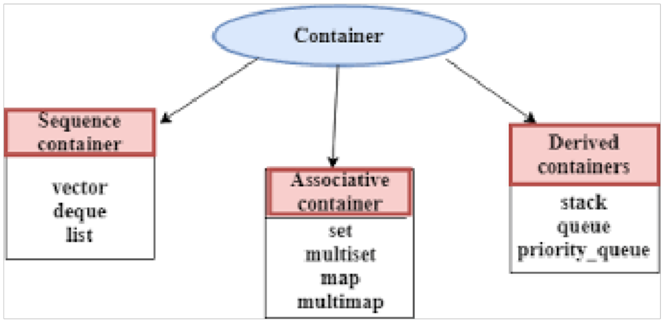
- A **container** is an object that actually stores data. It is a way in which data is organized in memory. The STL containers are implemented by template classes and therefore can be easily customized to hold different types of data.
- An **algorithm** is a procedure used to process the data contained in the containers. The STL includes many different kinds of algorithms to provide support to tasks such as initializing, searching, coping, sorting and merging. Algorithms are implemented by templates functions.
- An **iterator** is an object (like pointer) that points to an element in a container. We can use iterators to move through the contents of containers. It is handled just like pointer and can be incremented and decremented. Iterator connect algorithm with containers and play a key role in the manipulation of data stored in the containers.

Features of STL

It helps in saving time, efforts, load fast, high quality programming because STL provides well written and tested components, which can be reuse in our program to make our program more robust.

Containers

As we know containers is an object that actually stores data. The STL defines ten containers which are grouped in three categories as shown in fig.



1. **Sequence containers**

➢ Sequence containers stores elements in liner sequence.
➢ Elements of these containers can be accessed using an iterator.

The STL provides three types of sequence containers.

- Vector
- List
- Deque

2. **Associative containers**

Associative containers are designed to support direct access to element using keys. They are not sequential. There are four types of associative containers.

- Set
- Multiset
- Map
- Multimap

All these containers store data in a structure called a tree which facilitates fast searching, deletion and insertion. However, these are very slow for random access and inefficient for sorting.

3. **Derived containers**

The STL provides three derived containers namely stack, queue, and priority queue. These are also called containers adaptors.

Stack Queues and priority queues can be created from different sequence containers. The derived containers do not support iterators and therefore we cannot use them for data manipulation. However, they support two member functions **pop ()** and **push()** for implementing deleting and inserting operations.

Containers supported by STL

Container	Description	Header file	Iterator
Vector	A dynamic array. Allows insertions and deletions at back. Permits direct access to any element.	<vector>	Random access
List	A bidirectional, linear list. Allows insertions and deletions anywhere.	<list>	Bidirectional
Deque	A double-ended queue. Allows insertions and deletions at both ends. Permit direct access to any element.	<deque>	Random access
Set	An associate container for storing unique sets. Allows rapid lookup. (No duplicates are allowed)	<set>	Bidirectional
multiset	An associate container for storing non- unique sets. (Duplicates allowed)	<set>	Bidirectional
Map	An associate container for storing unique key/value pairs. Each key is associated with only one value (one-to-one mapping). Allows key-based lookup.	<map>	Bidirectional
multimap	An associate container for storing key/value pairs in which one key may be associated with more than one value. (one-to-many mapping) .Allows key-based lookup.	<map>	Bidirectional
Stack	An standard stack. Last-in-first-out (LIFO)	<stack>	No iterator
Queue	A standard queue. First-in-first-out (FIFO)	<queue>	No iterator
Priority-queue	A priority queues. The first element out is always the highest priority element.	<queue>	No iterator

Application of container classes

Vectors

The vector is the most widely used container. It stores elements in a contiguous memory location and enables direct access to any element using the subscript operator []. A vector can change its size dynamically and therefore allocates memory as needed at runtime.

The **vector** container supports random access iterators and wide range of iterator operations may be applied to a vector iterator. Class vector supports a number of constructors for creating vector objects.

```
vector<int> v1;           //Zero-length int vector
vector<double> v2(10); //10-element double vector
vector<int> v3(v4);      //creates v3 from v4
vector<int> v(5,2);        //5 element vector of 2s
```

The vector class supports several member functions as listed in Table below. we can also use all STL algorithms on a **vector.**

Function	Task
at()	Gives a reference to an element
back()	Gives a reference to the last element
begin()	Gives a reference to the first element
capacity()	Gives the current capacity of the vector
clear()	Deletes all the elements from the vector
empty()	Determines if the vector is empty or not
end()	Gives a reference to the end of the vector
erase()	Deletes specified elements
insert()	Inserts elements in the vector
pop_back()	Deletes the last element
push_back()	Adds an element to the end
resize()	modifies size of the vector to the specified value
size()	Gives the number of elements
swap()	Exchanges elements in the specified two vectors

Program below illustrates the use of several functions of the vector class template. Note that an iterator is used as pointer to elements of the vector. We must include header file <vector> to use vector class in our programs.

Write a C++ program to illustrate STL-`vector and its operations

```
#include<iostream>
#include<vector>
using namespace std;
void display(vector<int> v)
{
	for(int i=0;i<v.size();i++)
	{
		cout<<v[i]<<endl;
	}
}
int main()
{
	vector<int> v;
	int x;
	cout<<"Enter five integer values"<<endl;
	for(int i=0;i<5;i++)
	{
		cin>>x;
		v.push_back(x);
	}
```

```
	cout<<"size after adding values="<<v.size()<<endl;
	//v.size() returns the number of elements
	v.pop_back(); //Deletes the last element
	cout<<"current contents"<<endl;
	display(v);

	vector<int>::iterator itr=v.begin();
	itr=itr+2; //itr points to 3rd element
	v.insert(itr,2,10);
	cout<<"contents after insering"<<endl;
	display(v);

	v.erase(v.begin()+2);//Removes 3rd element
	cout<<"content after deletion"<<endl;
	display(v);
	cout<<"First value ="<<v.front();
	cout<<"Last value ="<<v.back();
	cout<<"value at index 2 ="<<v.at(2)<<endl;
	v.clear();//Deletes the all the elements from the vector
	cout<<"After clearing"<<endl;
	display(v);
	return 0;
}
```

Previous old Questions from this chapter

1) What do you mean by generic programming? Illustrate with the example of function template. [PU:2015 spring]
2) What are the advantages of Generic programming? Explain with suitable example.
3) What is function template?
4) What is template? List merit and demerit of using template in C++.
5) What is generic and templates. [PU:2016 spring]
6) What is template? List the merit and demerit of using a template in C++. [PU:2013 fall]
7) What is template? Explain different types of templates used in C++. [PU:2014 spring]
8) What are the advantages of using template functions. Write a program to illustrate a template function with two arguments. [PU:2017 fall]
9) With an example explain the concept of generic programming.
10) Explain the purpose of template programming with examples.
11) Write a short note on:

- Standard Template Library (STL)
- Template functions [PU:2019 fall]
- Template class [PU:2020 fall]

Programs

1. Write a program using template to add two integers, two floats and one integer and one float numbers respectively. Display the final result in float. [PU:2005 fall]
2. Write a function template to calculate the sum and average of numbers. [PU:2009 fall]
3. Create a template function to swap two values. [PU:2018 fall]
4. Write a function template to calculate the average and multiplication of numbers.
5. Create a template to find the sum of two integers and floats. [PU:2014 fall] .[PU:2016 spring] .[PU:2017 spring]
6. Define a class called stack and implement generic methods to push and pop the elements from the stack. [PU:2015 fall]
7. Write a program to illustrate the overloading of template functions.
8. How can we compute the roots of quadratic equations by using function template? Explain with examples.
9. Create a template function swap () and use it to swap two integers, two floating point data and two characters**. [PU:2020 fall]**

Define two classes named 'Polar' and 'rectangle' to represent points in polar and rectangle systems. Use conversion routines to convert from one system to another system using template.

```
#include <iostream>
#include <math.h>
using namespace std;

template<class T>
class Polar
{
private:
T radius;
T angle;
public:
Polar()
{
radius=0.0;
angle=0.0;
}
Polar(T r,T a)
{
radius=r;
angle=a;
}
void display()
{
cout<<"("<<radius<<","<<angle<<")"<<endl;
}
```

```
T getr()
{
return radius;
}
T geta()
{
return angle;
}
};

template<class T>
class Rectangle
{
private:
T xco;
T yco;
public:
Rectangle()
{
xco=0.0;
yco=0.0;
}
Rectangle(T x,T y)
{
        xco=x;
        yco=y;
}
void display()
{
cout<<"("<<xco<<","<<yco<<")"<<endl;
}
operator Polar<T>()
{
float a=atan(yco/xco);
float r=sqrt(xco*xco+yco*yco);
return Polar<T>(r,a);
}
Rectangle(Polar<T> p)
{
float r=p.getr();
float a=p.geta();
xco=r*cos(a);
yco=r*sin(a);
}
};
```

```
int main()
{
cout<<"Conversion from Polar to Rectangle coordinates"<<endl;
Polar<float> p1(10.0,0.785398);
Rectangle<float> r1;
r1=p1;
cout<<"Polar coordinates=";
p1.display();
cout<<"Rectangle coordinates=";
r1.display();
cout<<"Conversion from Rectangle to Polar coordinates"<<endl;
Rectangle<float> r(7.07107,7.07107);
Polar<float> p;
p=r;
cout<<"Rectangle coordinates=";
r.display();
cout<<"Polar coordinates=";
p.display();
return 0;
}
```