Templates and Generic Programming

Template is a new concept that enables us to define generic classes and functions and thus provides support for generic programming.

A template can be used to create family of classes or functions.for eg:a class templates for an array class would enable us to create arrays of various data types such as int array or float array or char array.

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.

Advantages and Disadvantages of using templates

Advantages:

- Class Template can handle different types of parameters.
- A template can be used to create family of classes or functions.
- Compiler generates classes for only the used types.
- Templates reduce the effort on coding for different data types to a single set of code.
- Testing and debugging efforts are reduced.

Disadvantages:

- Some compilers exhibited poor support for templates.So,the use of templates could decrease code portability.
- Because a template by its nature exposes its implementation,so its use in large systems can lead to longer build times.
- Templates are in the headers,which require a complete rebuild of all project pieces when changes are made.

Types of Template

There are basically two types of template.They are:

1. Function Template
2. Class Template

1)Function Template: It is used to create a family of functions with different argument types.for eg:we can define a template for a function,say mul() that would helps us to create various version of mul() for multiplying int,float and double type values.

a)Function template with single parameter:

General Format

```
template<class T>
return-type function-name(argument list of type T){
//body of the function with type T
}
```

Eg:WAP to calculate the sum of two integers and two float values using function template.

Ans:
```
#include<iostream>
using namespace std;
template<class T>
T add(T x,T y){

        return  (x+y);
}
main(){
        int a=2,d=3;
        float b=2.3,c=6.2;
        cout<<"sum of int and int values="<<add(a,d)<<endl;
```

```
    cout<<"sum of float and float values="<<add(b,c);
    return 0;
}
```

Q1)WAP to find the greatest among two integers and two float variable using function template.

Ans:
```
#include <iostream>
using namespace std;
template <class T>
T Large(T n1, T n2)
{
    return (n1 > n2) ? n1 : n2;
}
int main()
{
    int i1, i2;
    float f1, f2;
    cout << "Enter two integers:";
    cin >> i1 >> i2;
    cout <<" larger value among two integers."<< Large(i1, i2) << endl;
    cout << "Enter two floating-point numbers:";
    cin >> f1 >> f2;
    cout << " the  larger value among two float."<<Large(f1, f2)<< endl;
    return 0;
}
```

Q3)WAP to swap the values of two integers as a=100 and b=200 and two float values as c=100.2 and d=98.6 using function template.

Ans:
```
#include<iostream>
using namespace std;
template<class T>
void swapp(T &x,T  &y){
        T temp;
        temp=x;
        x=y;
        y=temp;
}
main(){
        int a=100,b=200;
        float c=100.2,d=98.6;
        swapp(a,b);
        cout<<"after swapping intergers value:"<<endl<<"a="<<a<<"b="<<b<<endl;
        swapp(c,d);
        cout<<"after swapping float values:"<<endl<<"c="<<c<<"d="<<d;
}
```

Q4)Write a function template to find average and multiplication of numbers.

Ans:
```
#include<iostream>
using namespace std;
template <class T>
void calculate(T x,T y){
        T s;
```

```
        s=(x+y)/2;
        cout<<"average of two numbers="<<s;

        T m;
        m=x*y;
        cout<<"multiplication of two numbers="<<m;
}
main(){
        int a=2, b=3;
        calculate(a,b);
}
```

Q4)WAP to find the maximum among the elements of an integer array having 6 elements and array of float having 4 elements using function template.

Ans:
```
#include<iostream>
using namespace std;
template<class T>
T min(T a[], int n)
{
   int i, j;
   T temp=a[0];
   for(i=0;i<n;i++)
   {
      if(a[i]>temp)
      temp=a[i];
   }
```

```
    return temp;
}
main()
{
   int min1;
    float min2;
    int a[6]={5,2,1,3,4,0};
    float b[4]={1.2,8.3,6.7,1.1};
   min1= min(a,6);
    cout<<"\nminimum of integer array: "<<min1;
    min2=min(b,4);
    cout<<"\nminimum of float array: "<<min2;
    return 0;
}
```

Bubble sorting using function template

```
#include<iostream>
using namespace std;
template<class T>
void bubble(T a[], int n)
{
    int i, j;
    for(i=0;i<n-1;i++)
    {
        for(j=i+1;j<n;j++)
```

```
        {
            if(a[i]>a[j])
            {
                T temp;
               temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
}

main()
{
     int i;
    int a[6]={5,2,1,3,4,0};
    float b[4]={1.2,8.3,6.7,1.1};
    bubble(a,6);
    cout<<"\nSorted Order Integers: ";
    for( i=0;i<6;i++)
        {
                    cout<<a[i]<<"\t";
          }
    bubble(b,4);
```

```
    cout<<"\nSorted Order floats: ";
    for( i=0;i<4;i++){
        cout<<b[i]<<"\t";
    }
    return 0;
}
```

b)Function template with multiple parameters(i.e. function with different generic types):

We can use more than one generic data type in template statement,using a comma-separated list.i.e.

General Format

```
template<class T,class T2,……..>
return-type function-name(argument of types T1,T2,….){
//body of function
}
```

Eg:WAP to find the sum of int,float values and float,int values.

Ans:
```
#include<iostream>
using namespace std;
template<class T1,class T2>
void add(T1 x,T2 y){
        cout<<"sum="<<x+y;
}
main(){
        int a=2,d=3;
        float b=2.3,c=6.2;
```

```
        cout<<"sum of int and float values="<<add(a,b);
        cout<<"sum of float and int values="<<add(c,d);
        return 0;
}
```

Q1)WAP to find the greatest among int ,float values and float,int values using function template.

Ans:
```
#include <iostream>
using namespace std;
template <class T1,class T2>
void Large(T1 n1, T2 n2)
{
        if(n1>n2)
         cout<<"largest is:"<<n1;
        else
          cout<<"largest is:"<<n2;
}
int main()
{
        int i1, i2;
        float f1, f2;
        cout << "Enter a integer and float value:";
        cin >>i1>>f1;
        Large(i1, f1);
        cout << "Enter a float and integer value:";
        cin >> f2 >> i2;
```

```
        Large(f2, i2);
        return 0;
}
```

***Use of **nesting of function template** is shown in the given example:

Eg:Bubble sorting using nesting of function template

```
#include<iostream>
using namespace std;
template<class T>
void bubble(T a[], int n)
{
   int i, j;
   for(i=0;i<n-1;i++)
   {
      for(j=i+1;j<n;j++)
      {
         if(a[i]>a[j])
         {
            swap(a[i],a[j]);
         }
      }
   }
}
 template<class X>
```

```
 void swap(X &a,X &b){
        X temp;
        temp=a;
        a=b;
        b=temp;
 }
main()
{
  int i;
   int a[6]={5,2,1,3,4,0};
   float b[4]={1.2,8.3,6.7,1.1};
   bubble(a,6);
   cout<<"\nSorted Order Integers: ";
   for( i=0;i<6;i++)
      cout<<a[i]<<"\t";
   bubble(b,4);
   cout<<"\nSorted Order floats: ";
   for( i=0;i<4;i++)
      cout<<b[i]<<"\t";
   return 0;
}
```

Overloading of template functions:

A function template can be overloaded either by template function or by the ordinary functions of its name.

An error is generated if no match is found.

Eg: #include<iostream>

```
using namespace std;
template<class T>
void display(T x){
        cout<<"value of template x="<<x;
}
template<class T,class T1>
void display(T x,T1 y){
        cout<<"value of x1="<<x<<"value of y1="<<y;
}
void display(int x){
        cout<<"value of normal function x="<<x;
}
main(){
        display(100);
        display(12.2);
        display(12,13.4);

}
```

Output:

Value of normal function x=100

Value of template function x=12.2

Value of x=12,y=13.4

2)Class Template

Class template with single parameter:

General Format:

template<class T>

class class-name{

//class member specifications with type T

}

The class template is very similar to the ordinary class definition except the prefix template<class T> and the use of T.This prefix tells the compiler that we are going to declare a template and use T as a typename in the declaration.

A class created from a class template is called a **template class.**The syntax for declaring an object of template class is:

Class-name<type>object-name;

This process of creating a specific object from a class template is known as **instantiation**.

Syntax for defining the member function of class template outside the class:

template<class T>

return-type class-name<T>::function-name(argumentlist with T){

//body of the function

}

Eg:WAP to calculate the sum of two integers and two float values using class template.

Ans: #include<iostream>

using namespace std;

template<class T>

class demo{

```
    T x,y;
    public:
        demo(T a,T b){
            x=a;
            y=b;
        }
        void display(){
            cout<<"sum="<<x+y;
        }
};
main(){
    demo<int>d1(1,2);
    d1.display();
    demo<float>d2(2.2,3.2);
    d2.display();
}
```

Q1)WAP to find out the greatest among the two integer values and two float values using class template.

Ans:
```
#include<iostream>
using namespace std;
template<class T>
class demo{
    T x,y;
    public:
```

```
        demo(T a,T b){
            x=a;
            y=b;
        }
        void display(){
            cout<<"greatest="<<(x>y?x:y);
        }
};
main(){
    demo<int>d1(1,2);
    d1.display();
    demo<float>d2(2.2,3.2);
    d2.display();
}
```

Q2)WAP to find the scalar product of two vectors in which both vector are of int type.

Ans: <u>First Method:</u>

```
#include<iostream>
using namespace std;
int size=3,i;
template<class T>
class vector{
    T *v;
    public:
        vector(){
```

```
        v=new T[size];
        for(i=0;i<size;i++){
            v[i]=0;
        }
    }
    vector(T a[]){
        for(i=0;i<size;i++){
            v[i]=a[i];
        }
    }
    T operator *(vector y){
        T sum=0;
        for(i=0;i<size;i++){
            sum+=v[i]*y.v[i];
        }
        return sum;
    }
    void display(){
        for(i=0;i<size;i++){
            cout<<v[i]<<" ";
        }
    }
};
main(){
```

```
    int x[3]={1,2,3};
    int y[3]={4,5,6};
    vector<int>v1;
    vector<int>v2;
    v1=x;
    v2=y;
    v1.display();
    v2.display();
    cout<<"scalar product of two vector i.e v1*v2="<<v1*v2;;

}
```

OR Second Method:

```
#include<iostream>
using namespace std;
template<class T>
class scalar{
    T a,b,c;
    public:
        scalar(){

        }
        scalar(T x,T y,T z){
            a=x;
            b=y;
```

```
            c=z;
        }
        T operator *(scalar p){
            T temp;
            temp=a*p.a+b*p.b+c*p.c;

            return temp;
        }
        void display(){
            cout<<a<<"i+"<<b<<"j+"<<c<<"k"<<endl;
        }
};
main(){
    int s3;
    scalar<int>s1(1,2,3);
    s1.display();
    scalar<int>s2(2,3,4);
    s2.display();
    s3=s1*s2;
    cout<<"scalar product="<<s3;
}
```

Q3)WAP to find the scalar product of two float type vectors using class template.

Ans: #include<iostream>

using namespace std;

```
int size=3,i;
template<class T>
class vector{
	T *v;
	public:
		vector(){
			v=new T[size];
			for(i=0;i<size;i++){
				v[i]=0;
			}
		}
		vector(T a[]){
				v=new T[size];
			for(i=0;i<size;i++){
				v[i]=a[i];
			}
		}
		T operator *(vector y){
			T sum=0;
			for(i=0;i<size;i++){
				sum+=v[i]*y.v[i];
			}
			return sum;
		}
```

```
        void display(){
            for(i=0;i<size;i++){
                cout<<v[i]<<" ";
            }
        }
};
main(){
    float x[3]={1,2,3};
    float y[3]={4,5,6};
    vector<float>v1;
    vector<float>v2;
    v1=x;
    v2=y;
    v1.display();
    v2.display();
    cout<<"scalar product of two vector i.e v1*v2="<<v1*v2;;
}
```

Q)A program to push and pop element in a stack using class template.

Ans:

```
#include<iostream>
using namespace std;
#define max 5
int i;
template<class T>
class stack{
```

```
T stk[max];
int top;
public:
        stack(){
                top=-1;
        }
        void push(T data){
                if(top==(max-1)){
                        cout<<"stack is full";
                }
                else{
                        top++;
                        stk[top]=data;
                }
        }
        void pop(){
                if(top==-1){
                        cout<<"stack is empty";
                }
                else{
                        top--;
                }
        }
        void show(){
```

```
                for(i=top;i>=0;i--){
                        cout<<" "<<stk[i]<<" ";
                }
        }
};
main(){
        stack<int>s;
        s.push(1);
        s.push(2);
        s.push(3);
        s.push(4);
        s.push(5);
        s.show();
        s.pop();
        s.show();
}
```

Class template with multiple parameters:

We can use more than one generic data types in a class template.

They are declared as comma-separated list within template specification as shown below:

```
template<class T1,class T2,…… >
class class-name{
//body of class
}
```

Eg:A program to find the sum of int,float and float,int values using class template.

Ans:

```
#include<iostream>
using namespace std;
template<class T1,class T2>
class demo{
        T1 a;
        T2 b;
        public:
                demo(T1 x,T2 y){
                        a=x;
                        b=y;
                }
                void display(){
                        cout<<"sum="<<a+b;
                }
};
main(){
int a,b;
float c,d;
cout<<"enter two integers:";
cin>>a>>b;
cout<<"enter two float values:";
cin>>c>>d;
demo<int,float>obj(a,c);
```

```
obj.display();
demo<float,int>obj1(d,b);
obj1.display();
}
```

Using default type in a class template

```
Eg: #include<iostream>
using namespace std;
template<class T1=int,class T2=int>
class demo{
        T1 a;
        T2 b;
        public:
                demo(T1 x,T2 y){
                        a=x;
                        b=y;
                }
                void display(){
                        cout<<"sum="<<a+b;
                }
};
main(){
int a=2,b=5;
float c=1.1,d=2.2;
```

```
demo<int,int>obj(a,c);
obj.display();
demo<float,int>obj1(d,b);
obj1.display();
demo<  >obj3(a,c);
obj3.display();
demo<float>obj4(c,a);
obj4.display();
}
```

Output:

```
Sum=3
Sum=7.2
Sum=3
Sum=3.1
```

Exception Handling

Problems other than the logic and syntax error is exception.**Exception** is defined as the run time anomalies or unusual conditions that a program may encounter while executing eg:division by zero,acces to an array outside of its bounds or running out of memory or disk space.

When a program encounters an exceptional conditions ,it is important that it is identified and dealt effectively.To deal with such anomalies/conditions special feature is used which is termed as exception.

Types of exception

There are two types of exception as listed below:

a)Synchronous exception:Errors such as “out-of range” index and overflow belongs to synchronous exception.

b)Asynchronous exception:Errors caused by events beyond the control of program such as keyboard interrupts belongs to asynchronous exception.

The purpose of exception handling mechanism is to provide means to detect and report an exceptional circumstance so that appropriate action can be taken.

Exception handling mechanism includes following steps:

1. Find the problem(Hit the exception)
2. Inform that the error has occurred(throw the exception)
3. Receive the error information(catch the exception)
4. Take corrective action(handle the exception)

Exception handling/Error handling mechanism contains two segments:

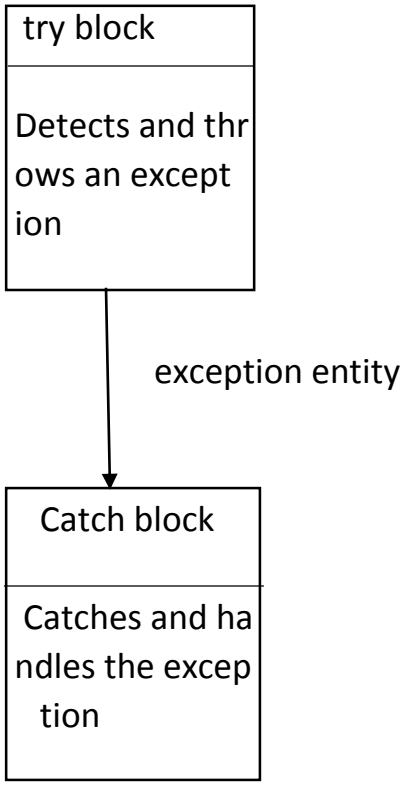
1. To detect errors and to throw the exceptions.
2. To catch the exceptions and take appropriate actions.

Exception Handling mechanism

This mechanism is basically built upon three keywords namely **try,throw** and **catch**.The block of statements that may generate the exceptions are included within the braces of try block.When an exception is detected it is thrown using a throw statement in the try block.

A catch block defined by the keyword catch catches the exception thrown by the throw statement in the try block and handles it properly.

The catch block that catches an exception must immediately follow the try block that throws the exceptions.



General form of try-catch block is:

try{
…….
throw (exception)
…….
…….}
catch(type argument){
……
…..}

When the try block throws an exception,the program control leaves the try block and enters the catch statement of the catch block.

Note:::

The exception are the objects used to transmit information about a problem.If the type of the object thrown matches the argument type in the catch statement then catch block is executed for handling the exception.If

they donot match,the program is aborted with the help of abort() function which is invoked by default(i.e. automatically by the compiler).

When the exception is detected and thrown ,the control goes to statement immediately after the catch block.

Eg:main(){
int a,b,x;
cout<<"enter a and b:";
cin>>a>>b;
x=a-b;
try{
if(x!=0)
 {
cout<<"result="<<a/x;
}
else{
throw(x);
}
}
catch(int i){
cout<<"exception caught:divide by zero";
}
}

Most often the exceptions are thrown by functions that are from within try blocks.The point at which throw is executed is called the throw point.

General format for this kind of exception is:

return-type function-name(argument list){
….
throw(object)
…
}
try{
…..

```
Call/invoke function here
}
catch(type argument){
….
Handles exception here
}
```

Eg:

```
void divide (int x,int y,int z)
{
if((x-y)!=0){
cout<<"result="<<z/(x-y);
}
else{
throw (x-y);
}
}
main(){
try{
divide(10,20,30);
divide(10,10,20);
}
catch(int i){
cout<<"caught an exception";
}
}
```

Multiple catch statement:

When a program segment has more than one condition to throw an exception,in such cases multiple catch statements with a try statement is used.**Syntax:**

```
try{
//try block
}
catch(type1 argument){
//catch block1
```

```
}
catch(type2 argument){
//catch block2
}
…….
…….
catch(typeN argument){
//catch blockN
}
```

When an exception is thrown the catch block is searched in order for an appropriate match.The first handler that yields a match is executed .After executing the handler block(i.e. catch),the control goes to the first statement after the last catch block for that try(in other words,all other handlers are skipped).When no match is found ,program is terminated.

Eg:

```
void test(int x){
 try{
  if(x==1)
        throw x;
 else if(x==0)
         throw 'x';
 else
          throw 1.2;
}
catch(int m){
cout<<"caught an integer";
}
Catch(char c){
```

```
cout<<"caught a character";
}
catch(float d){
cout<<"caught a float value";
}
}
main(){
cout<<"testing multiple catches:";
test(1);
test(0);
test(-1);
}
```

Rethrowing an exception

A catch block can also decide to rethrow the exception caught without processing it.In such situations we may simply invoke throw without any arguments as shown below:

```
void divide(float x,float y){
try{
If(y==0.0)
        throw y;
else
    cout<<"division:"<<x/y;
}
catch(float a){
```

```
cout<<"caught float ";
throw;//rethrows exception
}
}
main(){
try{
    divide(10.5,2.0);
    divide(20.0,0.0);
}
catch(float b){
cout<<"caught exception";
}
}
```

When an exception is rethrown ,it will not be caught by the same catch statement or any other catch in that group.Rather,it will be caught by an appropriate catch in the outer try/catch sequence only.

Standard Template Library(STL)

The collection of general-purpose templatized classes(data structures)and functions(algorithms)that could be used as a standard approach for storing and processing of data is known **as standard template library(STL).**

STL are defined under the directive **using namespace std.**

Components of STL

1. Containers
2. Algorithms
3. Iterators

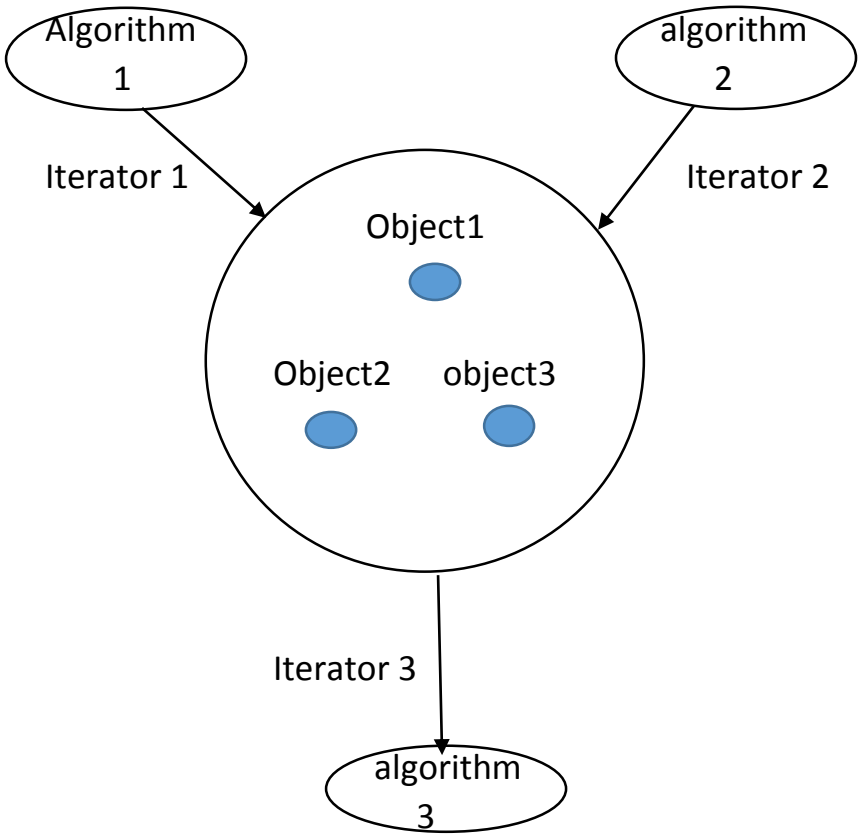


Fig:relationship between three STL components.

1)Containers:A container is an object that actually stores data.It is a way 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.The STL defines 10 containers which are grouped into 3 categories as given below:

- **Sequeunce containers:**This container stores elements in a linear sequence.Each element is related to other elements by its position along the line.Eg:vector,list,deque.
- **Associative containers:**These are designed to support direct access to elements using keys.They are not sequential .There are four types of associative containers they are set,multiset,map,multimap.All these containers stores data in a structure called tree which facilitate fast

searching,deletion and insertion.But these are very slow for random access and inefficient for storing.

- **Derived containers:**These are the containers derived from different sequence containers.These are also known as **container adaptors**.This donot support iterators and so we cannot use them for manipulation .It support two member function push() an pop() for deleting and inserting elements.There are three types of derived containers they are stack,queue,priority-queue.

2)Algorithms: Algorithms are functions that can be used generally across a variety of containers for processing their contents.STL provides more than 60 standard algorithms to support more extended or complex operations.It allows us to work with two different type of containers at the same time.To access standard algorithm we must include<algorithm>in our program.STL algorithms,based on the nature of operations they perform,it is categorized as:

Retrieve or nonmutating algorithms.

- Mutating algorithms
- Sorting algorithms
- Set algorithms
- Relational algorithms

3)Iterators: Iterators behave like pointers and are used to access container elements.They are often use to go from one element to another,a process known as iterating through the container.There are five types of iterators they are:input,output,forward,bidirectional,random.Each type of iterator is used for performing certain actions.The input and output supports the least functions.They can be used only to traverse in container.The forward iterator supports all the functions of input and output and also retains its position in the container.A bidirectional iterator,while supporting all forward iterator operations,provides the ability to move in the backward direction in the container.A random access iterator combines the functionality of a bidirectional iterator with the ability to jump to an arbitrary location.