Standard Template Library

By Anilabha Baral

C++ Templates

Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.

What are **generic functions or classes?**

Many times while programming, there is a need for creating functions which perform the same operations but work with different data types. So C++ provides a feature to create a single generic function instead of many functions which can work with different data type by using the **template parameter**.

What is the **template parameter?**

The way we use normal parameters to pass as a value to function, in the same manner template parameters can be used to pass type as argument to function. Basically, it tells what type of data is being passed to the function.

The syntax for creating a generic function:

```
template <class  type> return-type function-name (parameter-list)
```

Here, 'type' is just a placeholder used to store the data type when this function is used you can use any other name instead class is used to specify the generic type of template, alternatively typename can be used instead of it.

Let's try to understand it with an example:

Assume we have to swap two variables of int type and two of float type. Then, we will have to make two functions where one can swap int type variables and the other one can swap float type variables. But here if we use a generic function, then we can simply make one function and can swap both type of variables by passing their different type in the arguments. Let's implement this:

```
#include <iostream>
using namespace std ;
// creating a generic function 'swap (parameter-list)' using template :
template <class X>
void swap( X &a, X &b) {
      X tp;
      tp = a;
      a = b;
      b = tp;
```

```
    cout << " Swapped elements values of a and b are  " << a << " and  " << b << "
respectively " << endl;
}

int main( ) {
    int a = 10, b = 20 ;
    float c = 10.5, d = 20.5 ;
            swap(a , b);        // function swapping 'int' elements
    swap(c , d);              // function swapping 'float' elements
    return 0;
}
```

Output :

```
Swapped elements values of a and b are  20 and 10 respectively.
Swapped elements values of a and b are  20.5 and 10.5 respectively.
```

After creating the generic function, compiler will automatically generate correct code for the type of data used while executing the function.

C++ STL also has some containers (pre-build data structures) like vectors, iterators, pairs etc. These are all generic class which can be used to represent collection of any data type.

Iterator

An iterator is any object that, points to some element in a range of elements (such as an array or a container) and has the ability to iterate through those elements using a set of operators (with at least the increment (++) and dereference (*) operators).

A pointer is a form of an iterator. A pointer can point to elements in an array, and can iterate over them using the increment operator (++). There can be other types of iterators as well. For each container class, we can define iterator which can be used to iterate through all the elements of that container.

Example:

For Vector:

```
vector <int>::iterator  it;
```

For List:

```
list <int>::iterator it;
```

You will see the implementations using iterators in the topics explained below.

String

C++ provides a powerful alternative for the **char***. It is not a built-in data type, but is a container class in the Standard Template Library. String class provides different string manipulation functions like concatenation, find, replace etc. Let us see how to construct a string type.

```
string s0;                                  // s0 = “”
string s1(“Hello”);                         // s1 = “Hello”
string s2 (s1);                             // s2 = “Hello”
string s3 (s1, 1, 2);                       // s3 = “el”
string s4 ("Hello World", 5);               // s4 = “Hello”
string s5 (5, ‘*’);                         // s5 = “*****”
string s6 (s1.begin(), s1.begin()+3);       // s6 = “Hel”
```

Here are some member functions:

append(): Inserts additional characters at the end of the string (can also be done using ‘**+**’ or ‘**+=**’ operator). Its time complexity is O(N) where N is the size of the new string.

assign(): Assigns new string by replacing the previous value (can also be done using ‘**=**’ operator).

at(): Returns the character at a particular position (can also be done using ‘**[ ]**’ operator). Its time complexity is O(1).

begin(): Returns an iterator pointing to the first character. Its time complexity is O(1).

clear(): Erases all the contents of the string and assign an empty string (“”) of length zero. Its time complexity is O(1).

compare(): Compares the value of the string with the string passed in the parameter and returns an integer accordingly. Its time complexity is O(N + M) where N is the size of the first string and M is the size of the second string.

copy(): Copies the substring of the string in the string passed as parameter and returns the number of characters copied. Its time complexity is O(N) where N is the size of the copied string.

c_str(): Convert the string into C-style string (null terminated string) and returns the pointer to the C-style string. Its time complexity is O(1).

empty(): Returns a boolean value, true if the string is empty and false if the string is not empty. Its time complexity is O(1).

end(): Returns an iterator pointing to a position which is next to the last character. Its time complexity is O(1).

erase(): Deletes a substring of the string. Its time complexity is O(N) where N is the size of the new string.

find(): Searches the string and returns the first occurrence of the parameter in the string. Its time complexity is O(N) where N is the size of the string.

insert(): Inserts additional characters into the string at a particular position. Its time complexity is O(N) where N is the size of the new string.

length(): Returns the length of the string. Its time complexity is O(1).
replace(): Replaces the particular portion of the string. Its time complexity is O(N) where N is size of the new string.
resize(): Resize the string to the new length which can be less than or greater than the current length. Its time complexity is O(N) where N is the size of the new string.
size(): Returns the length of the string. Its time complexity is O(1).
substr(): Returns a string which is the copy of the substring. Its time complexity is O(N) where N is the size of the substring.

Implementation:

```
#include <iostream>
#include <cstdio>

using namespace std;

int main()
{
	string s, s1;
	s = "HELLO";
	s1 = "HELLO";
	if(s.compare(s1) == 0)
		cout << s << " is equal to " << s1 << endl;
	else
		cout << s << " is not equal to " << s1 << endl;
	s.append(" WORLD!");
	cout << s << endl;
	printf("%s\n", s.c_str());
	if(s.compare(s1) == 0)
		cout << s << " is equal to " << s1 << endl;
	else
		cout << s << " is not equal to " << s1 << endl;
	return 0;
}
```

Output:

```
HELLO is equal to HELLO
HELLO WORLD!
HELLO WORLD!
HELLO WORLD! is not equal to HELLO
```

Vector

Vectors are sequence containers that have dynamic size. In other words, vectors are dynamic arrays. Just like arrays, vector elements are placed in contiguous storage location so they can be accessed and traversed using iterators. To traverse the vector we need the position of the first and last element in the vector which we can get through **begin()** and **end()** or we can use indexing from 0 to **size()**. Let us see how to construct a vector.

```
vector<int> a;                          // empty vector of ints
vector<int> b (5, 10);                     // five ints with value 10
vector<int> c (b.begin(),b.end());                 // iterating through second
vector<int> d (c);                       // copy of c
```

Some of the member functions of vectors are:

at(): Returns the reference to the element at a particular position (can also be done using '[ ]' operator). Its time complexity is O(1).

back(): Returns the reference to the last element. Its time complexity is O(1).

begin(): Returns an iterator pointing to the first element of the vector. Its time complexity is O(1).

clear(): Deletes all the elements from the vector and assign an empty vector. Its time complexity is O(N) where N is the size of the vector.

empty(): Returns a boolean value, true if the vector is empty and false if the vector is not empty. Its time complexity is O(1).

end(): Returns an iterator pointing to a position which is next to the last element of the vector. Its time complexity is O(1).

erase(): Deletes a single element or a range of elements. Its time complexity is O(N + M) where N is the number of the elements erased and M is the number of the elements moved.

front(): Returns the reference to the first element. Its time complexity is O(1).

insert(): Inserts new elements into the vector at a particular position. ts time complexity is O(N + M) where N is the number of elements inserted and M is the number of the elements moved .

pop_back(): Removes the last element from the vector. Its time complexity is O(1).

push_back(): Inserts a new element at the end of the vector. Its time complexity is O(1).

resize(): Resizes the vector to the new length which can be less than or greater than the current length. Its time complexity is O(N) where N is the size of the resized vector.

size(): Returns the number of elements in the vector. Its time complexity is O(1).

Traverse:

```
void traverse(vector<int> v)
{
```

```
    vector <int>::iterator it;
    for(it = v.begin();it != v.end();++it)
        cout << *it <<  ' ';
    cout << endl;
    for(int i = 0;i < v.size();++i)
        cout << v[i] << ' ';
    cout << endl;
 }
```

Implementation:

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector <int> v;
    vector <int>::iterator it;
    v.push_back(5);
    while(v.back() > 0)
        v.push_back(v.back() - 1);
    for(it = v.begin(); it != v.end();++it)
        cout << *it << ' ';
    cout << endl;
    for(int i = 0;i < v.size();++i)
        cout << v.at(i) << ' ';
    cout << endl;
    while(!v.empty())
    {
        cout << v.back() << ' ';
        v.pop_back();
    }
    cout << endl;
    return 0;
}
```

Output:

```
5 4 3 2 1 0
```

```
5 4 3 2 1 0
0 1 2 3 4 5
```

List
List is a sequence container which takes constant time in inserting and removing elements. List in STL is implemented as Doubly Link List.

The elements from List cannot be directly accessed. For example to access element of a particular position ,you have to iterate from a known position to that particular position.

//declaration

```
list <int> LI;
```

Here LI can store elements of int type.

// here LI will have 5 int elements of value 100.

```
list<int> LI(5, 100)
```

Some of the member function of List:
begin( ): It returns an iterator pointing to the first element in list.Its time complexity is O(1).
end( ): It returns an iterator referring to the theoretical element(doesn't point to an element) which follows the last element.Its time complexity is O(1).
empty( ): It returns whether the list is empty or not.It returns 1 if the list is empty otherwise returns 0.Its time complexity is O(1).
assign( ): It assigns new elements to the list by replacing its current elements and change its size accordingly.It time complexity is O(N).
back( ): It returns reference to the last element in the list.Its time complexity is O(1).
erase( ): It removes a single element or the range of element from the list.Its time complexity is O(N).
front( ): It returns reference to the first element in the list.Its time complexity is O(1).
push_back( ): It adds a new element at the end of the list, after its current last element. Its time complexity is O(1).
push_front( ): It adds a new element at the beginning of the list, before its current first element. Its time complexity is O(1).
remove( ): It removes all the elements from the list, which are equal to given element. Its time complexity is O(N).
pop_back( ): It removes the last element of the list, thus reducing its size by 1. Its time complexity is O(1).
pop_front( ): It removes the first element of the list, thus reducing its size by 1. Its time complexity is O(1).

insert( ): It insert new elements in the list before the element on the specified position. Its time complexity is O(N).
reverse ( ): It reverses the order of elements in the list. Its time complexity is O(N).
size( ): It returns the number of elements in the list. Its time complexity is O(1).

Implementation:

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
	list <int> LI;
	list <int>::iterator it;
	//inserts elements at end of list
	LI.push_back(4);
	LI.push_back(5);

	//inserts elements at beginning of list
	LI.push_front(3);
	LI.push_front(5);

	//returns reference to first element of list
	it = LI.begin();

	//inserts 1 before first element of list
	LI.insert(it,1);

	cout<<"All elements of List LI are: " <<endl;
	for(it = LI.begin();it!=LI.end();it++)
	{
		cout<<*it<<" ";
	}
	cout<<endl;

	//reverse elements of list
	LI.reverse();

	cout<<"All elements of List LI are after reversing: " <<endl;
	for(it = LI.begin();it!=LI.end();it++)
	{
		cout<<*it<<" ";
```

```
    }
    cout<<endl;

    //removes all occurences of 5 from list
    LI.remove(5);

    cout<<"Elements after removing all occurence of 5 from List"<<endl;
    for(it = LI.begin();it!=LI.end();it++)
    {
        cout<<*it<<" ";
    }
    cout<<endl;

    //removes last element from list
    LI.pop_back();
    //removes first element from list
    LI.pop_front();
    return 0;
}
```

Output:

```
All elements of List LI are:
1 5 3 4 5
All elements of List LI are after reversing:
5 4 3 5 1
Elements after removing all occurrence of 5  from List
4 3 1
```

Pair

Pair is a container that can be used to bind together a two values which may be of different types. Pair provides a way to store two heterogeneous objects as a single unit.

```
pair <int, char> p1;              // default
pair <int, char> p2 (1, 'a');       // value inititialization
pair <int, char> p3 (p2);          // copy of p2
```

We can also initialize a pair using **make_pair()** function. **make_pair(x, y)** will return a pair with first element set to x and second element set to y.

```
p1 = make_pair(2, 'b');
```

To access the elements we use keywords, first and second to access the first and second element respectively.

```
cout << p2.first << ' ' << p2.second << endl;
```

Implementation:

```
#include <iostream>
#include <utility>

using namespace std;

int main()
{
	pair <int, char> p;
	pair <int, char> p1(2, 'b');
	p = make_pair(1, 'a');
	cout << p.first << ' ' <<  p.second << endl;
	cout << p1.first << ' ' << p1.second << endl;
	return 0;
}
```

Output:

```
1 a
2 b
```

Sets

Sets are containers which store only unique values and permit easy look ups. The values in the sets are stored in some specific order (like ascending or descending). Elements can only be inserted or deleted, but cannot be modified. We can access and traverse set elements using iterators just like vectors.

```
set<int> s1;                        // Empty Set
int a[]= {1, 2, 3, 4, 5, 5};
set<int> s2 (a, a + 6);                 // s2 = {1, 2, 3, 4, 5}
set<int> s3 (s2);                     // Copy of s2
set<int> s4 (s3.begin(), s3.end());       // Set created using iterators
```

Some of the member functions of set are:

begin(): Returns an iterator to the first element of the set. Its time complexity is O(1).

clear(): Deletes all the elements in the set and the set will be empty. Its time complexity is O(N) where N is the size of the set.
count(): Returns 1 or 0 if the element is in the set or not respectively. Its time complexity is O(logN) where N is the size of the set.
empty(): Returns true if the set is empty and false if the set has at least one element. Its time complexity is O(1).
end(): Returns an iterator pointing to a position which is next to the last element. Its time complexity is O(1).
erase(): Deletes a particular element or a range of elements from the set. Its time complexity is O(N) where N is the number of element deleted.
find(): Searches for a particular element and returns the iterator pointing to the element if the element is found otherwise it will return the iterator returned by end(). Its time complexity is O(logN) where N is the size of the set.
insert(): insert a new element. Its time complexity is O(logN) where N is the size of the set.
size(): Returns the size of the set or the number of elements in the set. Its time complexity is O(1).

Traverse:

```
void traverse(set<int> s)
{
      set <int>::iterator it;
      for(it = s.begin();it != s.end();++it)
            cout << *it <<  ' ';
      cout << endl;
 }
```

Implementation:

```
#include <iostream>
#include <set>

using namespace std;

int main()
{
      set <int> s;
      set <int>::iterator it;
      int A[] = {3, 5, 2, 1, 5, 4};
      for(int i = 0;i < 6;++i)
            s.insert(A[i]);
```

```
    for(it = s.begin();it != s.end();++it)
        cout << *it << ' ';
    cout << endl;
    return 0;
}
```

Output:

```
1 2 3 4 5
```

Maps

Maps are containers which store elements by mapping their value against a particular key. It stores the combination of key value and mapped value following a specific order. Here key value are used to uniquely identify the elements mapped to it. The data type of key value and mapped value can be different. Elements in map are always in sorted order by their corresponding key and can be accessed directly by their key using bracket operator ([ ]).

In map, key and mapped value have a pair type combination,i.e both key and mapped value can be accessed using pair type functionalities with the help of iterators.

//declaration. Here key values are of char type and mapped values(value of element) is of int type.

```
map <char ,int > mp;

mp['b']  = 1;
```

It will map value 1 with key 'b'. We can directly access 1 by using mp['b'].

```
mp['a'] = 2;
```

It will map value 2 with key 'a'.

In map mp , the values be will be in sorted order according to the key.
Note: N is the number of elements in map.
Some Member Functions of map:
at( ): Returns a reference to the mapped value of the element identified with key.Its time complexity is O(logN).
count( ): searches the map for the elements mapped by the given key and returns the number of matches.As map stores each element with unique key, then it will return 1 if match if found otherwise return 0.Its time complexity is O(logN).
clear( ): clears the map, by removing all the elements from the map and leaving it with its

size 0.Its time complexity is O(N).
begin( ): returns an iterator(explained above) referring to the first element of map.Its time complexity is O(1).
end( ): returns an iterator referring to the theoretical element(doesn't point to an element) which follows the last element.Its time complexity is O(1).
empty( ): checks whether the map is empty or not. It doesn't modify the map.It returns 1 if the map is empty otherwise returns 0.Its time complexity is O(1).
erase( ): removes a single element or the range of element from the map.
find( ): Searches the map for the element with the given key, and returns an iterator to it, if it is present in the map otherwise it returns an iterator to the theoretical element which follows the last element of map.Its time complexity is O(logN).
insert( ): insert a single element or the range of element in the map.Its time complexity is O(logN), when only element is inserted and O(1) when position is also given.

Implementation:

```
#include <iostream>
#include <map>
using namespace std;
int main(){
	map <char,int> mp;
	map <char,int> mymap,mymap1;

	//insert elements individually in map with the combination of key value and value of element
	mp.insert(pair<char,int>('a',2));   //key is 'c' and 2 is value.
	mp.insert(pair<char,int>('b',1));
	mp.insert(pair<char,int>('c',43));

	//inserts elements in range using insert() function in map 'mymap'.
	mymap.insert(mp.begin(),mp.end());

	//declaring iterator for map
	map <char,int>::iterator it;

	//using find() function to return reference of element mapped by key 'b'.
	it = mp.find('b');

	//prints key and element's value.
	cout<<"Key and element's value of map are: ";
	cout<<it->first<<" and "<<it->second<<endl;
```

```
	//alternative way to insert elements by mapping with their keys.
	mymap1['x'] = 23;
	mymap1['y'] = 21;

	cout<<"Printing element mapped by key 'b' using at() function :
"<<mp.at('b')<<endl;

	//swap contents of 2 maps namely mymap and mymap1.
	mymap.swap(mymap1);

	/* prints swapped elements of mymap and mymap1 by iterating all the elements
through
		using	iterator. */
	cout<<"Swapped elements and their keys of mymap are: "<<endl;
	for(it=mymap.begin();it!=mymap.end();it++)
	{
	cout<<it->first<<" "<<it->second<<endl;
	}
	cout<<"Swapped elements and their keys of mymap1 are: "<<endl;
	for(it=mymap1.begin();it!=mymap1.end();it++)
	{
	cout<<it->first<<" "<<it->second<<endl;
	}
	//erases element mapped at 'c'.
	mymap1.erase('c');

	//prints all elements of mymap after erasing element at 'c'.

	cout<<"Elements of mymap1 after erasing element at key 'c' : "<<endl;
	for(it=mymap1.begin();it!=mymap1.end();it++)
	{
	cout<<it->first<<" "<<it->second<<endl;
	}

	//erases elements in range from mymap1
	mymap1.erase(mymap1.begin(),mymap1.end());

	cout<<"As mymap1 is empty so empty() function will return 1 : " <<
mymap1.empty()<<endl;

	//number of elements with key = 'a' in map mp.
```

```
	cout<<"Number of elements with key = 'a' in map mp are :
"<<mp.count('a')<<endl;

	//if mp is empty then itmp.empty will return 1 else 0.
	if(mp.empty())
	{
		cout<<"Map is empty"<<endl;
	}
	else
	{
		cout<<"Map is not empty"<<endl;
	}

	return 0;
}
```

Output:

```
Key and element's value of map are: b and 1
Printing element mapped by key 'b' using at() function : 1
Swapped elements and their keys of mymap are:
x 23
y 21
Swapped elements and their keys of mymap1 are:
a 2
b 1
c 43
Elements of mymap1 after erasing element at key 'c' :
a 2
b 1
As mymap1 is empty so empty() function will return 1 : 1
Number of elements with key = 'a' in map mp are : 1
Map is not empty
```

Stacks:

Stack is a container which follows the **LIFO (Last In First Out)** order and the elements are inserted and deleted from one end of the container. The element which is inserted last will be extracted first.

Declaration:

```
stack <int> s;
```

Some of the member functions of Stack are:
push( ): Insert element at the top of stack. Its time complexity is O(1).
pop( ): removes element from top of stack. Its time complexity is O(1).
top( ): access the top element of stack. Its time complexity is O(1).
empty( ): checks if the stack is empty or not. Its time complexity is O(1).
size( ): returns the size of stack. Its time complexity is O(1).

Implementation:

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

using namespace std;
int main( )
{
	stack <int> s;  // declaration of stack

	//inserting 5 elements in stack from 0 to 4.
	for(int i = 0;i < 5; i++)
	{
		s.push( i ) ;
	}

	//Now the stack is {0, 1, 2, 3, 4}

	//size of stack s
	cout<<"Size of stack is: " <<s.size( )<<endl;

	//accessing top element from stack, it will be the last inserted element.
	cout<<"Top element of stack is: " <<s.top( ) <<endl ;

	//Now deleting all elements from stack
	for(int i = 0;i < 5;i++)
	{
		s.pop( );
	}

	//Now stack is empty,so empty( ) function will return true.
```

```
    if(s.empty())
    {
        cout <<"Stack is empty."<<endl;
    }
    else
    {
        cout <<"Stack is Not empty."<<endl;
    }

    return 0;

}
```

Output:

```
Size of stack is: 5
Top element of stack is: 4
Stack is empty.
```

Queues:
Queue is a container which follows **FIFO order (First In First Out)** . Here elements are inserted at one end (rear) and extracted from another end(front) .

Declaration:

```
queue <int> q;
```

Some member function of Queues are:
push( ): inserts an element in queue at one end(rear). Its time complexity is O(1).
pop( ): deletes an element from another end if queue(front). Its time complexity is O(1).
front( ): access the element on the front end of queue. Its time complexity is O(1).
empty( ): checks if the queue is empty or not. Its time complexity is O(1).
size( ): returns the size of queue. Its time complexity is O(1).

Implementation:

```
#include <iostream>
#include <cstdio>
#include <queue>

using namespace std;
```

```
int main() {
	char qu[4] = {'a', 'b', 'c', 'd'};
	queue <char> q;
	int N = 3;			// Number of steps
	char ch;
	for(int i = 0;i < 4;++i)
		q.push(qu[i]);
	for(int i = 0;i < N;++i) {
			ch = q.front();
	q.push(ch);
			q.pop();
	}
	while(!q.empty()) {
		printf("%c", q.front());
		q.pop();
	}
	printf("\n");
	return 0;
}
```

Output:

```
dabc
```

Priority Queue:
A priority queue is a container that provides constant time extraction of the largest element, at the expense of logarithmic insertion. It is similar to the heap in which we can add element at any time but only the maximum element can be retrieved. In a priority queue, an element with high priority is served before an element with low priority.

Declaration:

```
priority_queue<int> pq;
```

Some member functions of priority queues are:
empty(): Returns true if the priority queue is empty and false if the priority queue has at least one element. Its time complexity is O(1).
pop(): Removes the largest element from the priority queue. Its time complexity is O(logN) where N is the size of the priority queue.
push(): Inserts a new element in the priority queue. Its time complexity is O(logN) where N is the size of the priority queue.

size(): Returns the number of element in the priority queue. Its time complexity is O(1).
top(): Returns a reference to the largest element in the priority queue. Its time complexity is O(1).

Implementation:

```
#include <iostream>
#include <queue>

using namespace std;

int main()
{
	priority_queue<int> pq;
	pq.push(10);
	pq.push(20);
	pq.push(5);
	while(!pq.empty())
	{
		cout << pq.top() << endl;
		pq.pop();
	}
	return 0;
}
```

Output:

```
20
10
5
```

AUTHOR ANILABHA BARAL