Object-oriented Programming
Templates and Introduction to STL

Templates

Template is a new concept in C++ which enables us to define generic functions and classes and thus provide supports for generic programming. A template can be used to create a family of functions or classes. For example we can use the same function or class for performing different type of data processing. Templates are sometimes called parameterized classes or functions.

Function templates

Function templates are special functions that can operate with *generic types*. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

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.
The format for declaring function templates with type parameters is:

template <class identifier> function_declaration;
template <typename identifier> function_declaration;

The only difference between both prototypes is the use of either the keyword class or the keyword typename. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

For example, to create a template function that returns the greater one of two objects we could use:

```
template <class myType>
myType GetMax (myType a, myType b) {
 return (a>b?a:b);
}
```

Here we have created a template function with myType as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type. As you can see, the function template GetMax returns the greater of two parameters of this still-undefined type.

To use this function template we use the following format for the function call:

function_name <type> (parameters);

For example, to call GetMax to compare two integer values of type int we can write:

```
int x,y;
GetMax <int> (x,y);
```

When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of myType by the type passed as the actual template parameter (int in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.

Here is the entire example:

```
// function template
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a>b)? a : b;
  return (result);
}

int main () {
  int i=5, j=6, k;
  double l=10.5, m=5.6, n;
  k=GetMax<int>(i,j);
  n=GetMax<double>(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

```
output
6
10.5
```

In this case, we have used T as the template parameter name instead of myType because it is shorter and in fact is a very common template parameter name. But you can use any identifier you like.

In the example above we used the function template GetMax() twice. The first time with arguments of type int and the second one with arguments of type double. The compiler has instantiated and then called each time the appropriate version of the function.

As you can see, the type T is used within the GetMax() template function even to declare new objects of that type:

```
T result;
```

Therefore, result will be an object of the same type as the parameters a and b when the function template is instantiated with a specific type.

In this specific case where the generic type T is used as a parameter for GetMax the compiler can find out automatically which data type has to instantiate without having to explicitly specify it within angle brackets (like we have done before specifying <int> and <double>). So we could have written instead:

```
int i,j;
GetMax (i,j);
```

Since both i and j are of type int, and the compiler can automatically find out that the template parameter can only be int. This implicit method produces exactly the same result:

```
// function template II
#include <iostream>
using namespace std;

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

int main () {
  int i=5, j=6, k;
  double l=10.5, m=5.6, n;
  k=GetMax(i,j);
  n=GetMax(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

```
output
6
10.5
```

Notice how in this case, we called our function template GetMax() without explicitly specifying the type between angle-brackets <>. The compiler automatically determines what type is needed on each call.

Because our template function includes only one template parameter (class T) and the function template itself accepts two parameters, both of this T type, we cannot call our function template with two objects of different types as arguments:

```
int i;
double l;
k = GetMax (i,l);
```

This would not be correct, since our GetMax function template expects two arguments of the same type, and in this call to it we use objects of two different types.

We can also define function templates that accept more than one type parameter, simply by specifying more template parameters between the angle brackets. For example:

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

In this case, our function template GetMin() accepts two parameters of different types and returns an object of the same type as the first parameter (T) that is passed. For example, after that declaration we could call GetMin()with:

```
int i,j;
double l;
i = GetMin<int,double> (j,l);
```

or simply:

```
i = GetMin (j,l);
```

even though j and l have different types, since the compiler can determine the appropriate instantiation anyway.

Class templates

We also have the possibility to write class templates, so that a class can have members that use template parameters as types. For example:

```
template <class T>
class mypair {
    T values [2];
  public:
    mypair (T first, T second)
    {
      values[0]=first; values[1]=second;
    }
};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type int with the values 115 and 36 we would write:

```
mypair<int> myobject (115, 36);
```

this same class would also be used to create an object to store any other type:

```
mypair<double> myfloats (3.0, 2.18);
```

The only member function in the previous class template has been defined inline within the class declaration itself. In case that we define a function member outside the declaration of the class template, we must always precede that definition with the template <...> prefix:

```
// class templates
#include <iostream>
using namespace std;

template <class T>
class mypair {
    T a, b;
  public:
    mypair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};

template <class T>
T mypair<T>::getmax ()
{
  T retval;
  retval = a>b? a : b;
  return retval;
}
```

```
output
100
```

```
int main () {
  mypair <int> myobject (100, 75);
  cout << myobject.getmax();
  return 0;
}
```

Notice the syntax of the definition of member function getmax:

```
template <class T>
T mypair<T>::getmax ()
```

Confused by so many T's? There are three T's in this declaration: The first one is the template parameter. The second T refers to the type returned by the function. And the third T (the one between angle brackets) is also a requirement: It specifies that this function's template parameter is also the class template parameter.

The Standard Template Library (STL)

One thing you've probably noticed is that an awful lot of programs use the same concepts over and over again: loops, strings, arrays, sorting, etc… You've probably also noticed that writing programs using non-class versions of containers and common algorithms are error-prone. The good news is that C++ comes with a library that is chock full of reusable classes for you to build programs out of. This library is called The C++ Standard Template Library (STL).

The Standard Template Library is a collection of classes that provide templated containers, algorithms, and iterators. If you need a common class or algorithm, odds are the STL has it. The upside is that you can take advantage of these classes without having to write and debug the classes yourself, and the STL does a good job providing reasonably efficient versions of these classes. The downside is that the STL is complex, and can be a little intimidating since everything is templated.

Fortunately, you can bite off the STL in tiny pieces, using only what you need from it, and ignore the rest until you're ready to tackle it.

In the next few lessons, we'll take a high-level look at the types of containers, algorithms, and iterators that the STL provides. **These STL components are part of C++ Library are defined in the *namespace std***.

STL containers

By far the most commonly used functionality of the STL library are the STL container classes. The STL contains many different container classes that can be used in different situations.

Generally speaking, the container classes fall into **three basic categories**: **Sequence containers**, **Associative containers**, and **Container adapters**. We'll just do a quick overview of the containers here.

Sequence containers:

- **vector:** Dynamic contiguous array (class template)
- **deque:** Double-ended queue (class template)
- **list :** Doubly-linked list (class template)

Associative containers

- **Set:** Collection of unique keys, sorted by keys
  (class template)
- **Map:** Collection of key-value pairs, sorted by keys, keys are unique (class template).
- **Multiset:** Collection of keys, sorted by keys (class template)

Container adaptors

- **stack:** Adapts a container to provide stack (LIFO data structure) (class template).
- **queue:** Adapts a container to provide queue (FIFO data structure) (class template).
- **priority_queue:** Adapts a container to provide priority queue (class template).

Sequence Containers

Sequence contains are container classes that maintain the ordering of elements in the container. A defining characteristic of sequence containers is that you can choose where to insert your element by position. The most common example of a sequence container is the array: if you insert four elements into an array, the elements will be in the exact order you inserted them.
The named **vector** class in the STL is a dynamic array capable of growing as needed to contain its elements. The vector class allows random access to its elements via operator [ ], and inserting and removing elements from the end of the vector is generally fast.
The following program inserts 6 numbers into a vector and uses the overloaded [] operator to access them in order to print them.

```
#include <vector>
#include <iostream>
int main()
{
    using namespace std;

    vector<int> vect, newVect;
    for (int nCount=0; nCount < 6; nCount++)
        vect.push_back(10 - nCount); // insert at end of array
    newVect = vect; //direct copy
    for (int nIndex=0; nIndex < vect.size(); nIndex++)
        cout << newVect[nIndex] << " ";

    cout << endl;
}
```

This program produces the result:
10 9 8 7 6 5

```
// comparing size, capacity and max_size
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;

  // set some content in the vector:
  for (int i=0; i<100; i++) myvector.push_back(i);

  std::cout << "size: " << myvector.size() << "\n";
  std::cout << "capacity: " << myvector.capacity() << "\n";
  std::cout << "max_size: " << myvector.max_size() << "\n";
  return 0;
}
```

```
size: 100
capacity: 128
max_size:
1073741823
```

2D Vector:

```
vector< int > v[100]; //2D

vector< vector< int > > v; //2D
```

deque

The **deque** class (pronounced “deck”) is a double-ended queue class. It is Functionality similar to vectors but with efficient insertion and deletion of elements also at the beginning of the sequence, and not only at its end.

```
#include <iostream>
#include <deque>
int main()
{
    using namespace std;

    deque<int> deq;
    for (int nCount=0; nCount < 3; nCount++)
    {
        deq.push_back(nCount); // insert at end of array
        deq.push_front(10 - nCount); // insert at front of array
    }

    for (int nIndex=0; nIndex < deq.size(); nIndex++)
```

```
        cout << deq[nIndex] << " ";

    cout << endl;
}
```

This program produces the result:
8 9 10 0 1 2

Stack Operations

```
#include <iostream>
#include <stack>
using namespace std;
int main()
{
    stack<int> s;

    s.push(8);
    s.push(5);
    s.push(6);

    cout << s.top() << endl;
    s.pop();
    cout << s.top() << endl;
}
```

```
Output:
6
5
```

Queue Operations

```
#include <iostream>
#include <queue>
using namespace std;
int main()
{
    queue<int> q;

    q.push(8);
    q.push(5);
    q.push(6);

    cout << q.front() << endl;
    q.pop();
    cout << q.front() << endl;
}
```

```
Output:
8
5
```

Priority queue

```
#include <iostream>
#include <queue>
using namespace std;
int main()
```

```
Output:
102322
10230
1021
```

```
{
    priority_queue<int> q;
    q.push( 10230 ); // inserting 10230
    q.push( 1021 ); // inserting 1021
    q.push( 102322 ); // inserting 102322

    while( !q.empty() ) {
        cout << q.top() << endl; // printing the top
        q.pop(); // removing that one
    }

}
```

Map

```
#include<iostream>
#include<map>
using namespace std;
int main()
{
    map<string,int> Mark;
    Mark["Allen"] = 95;
    Mark["Edward"] = 87;
    Mark["Louise"] = 66;

    map<string,int> :: iterator it;

    for(it=Mark.begin(); it!=Mark.end(); it++)
    {
        cout << it->first << '\t' << it->second << '\n';
    }

    cout << Mark["Allen"] << endl;
}
```

```
Output:
Allen 95
Edward 87
Louise 66
95
```

Common STL algorithms:

Some of the most used algorithms on vectors and most useful one's in Competitive Programming are mentioned as follows:

- **sort(first_iterator, last_iterator)** – To sort the given vector.
- **reverse(first_iterator, last_iterator)** – To reverse a vector.
- ***max_element (first_iterator, last_iterator)** – To find the maximum element of a vector.

- ❖ ***min_element (first_iterator, last_iterator)** – To find the minimum element of a vector.

- ❖ **accumulate(first_iterator, last_iterator, initial value of sum)** – Does the summation of vector elements

- ❖ **count(first_iterator, last_iterator, x)** – To count the occurrences of x in vector.

- ❖ **find(first_iterator, last_iterator, x)** – Points to last address of vector ((name_of_vector).end()) if element is not present in vector.

- ❖ **binary_search(first_iterator, last_iterator, x)** – Tests whether x exists in sorted vector or not.

- ❖ **lower_bound(first_iterator, last_iterator, x)** – returns an iterator pointing to the first element in the range [first,last) which has a value not less than ‘x’.

- ❖ **upper_bound(first_iterator, last_iterator, x)** – returns an iterator pointing to the first element in the range [first,last) which has a value greater than ‘x’.

- ❖ **arr.erase(position to be deleted)** – This erases selected element in vector and shifts and resizes the vector elements accordingly.

- ❖ **arr.erase(unique(arr.begin(),arr.end()),arr.end())** – This erases the duplicate occurrences in sorted vector in a single line.

- ❖ **distance(first_iterator,desired_position)** – It returns the distance of desired position from the first iterator.This function is very useful while finding the index.

```
cout << distance(vect.begin(),
max_element(vect.begin(), vect.end()));
```

References:

1. https://sites.google.com/site/smilitude/stl (Fahim er Blog)
2. http://www.cplusplus.com/ /reference/stl/
3. https://www.geeksforgeeks.org/cpp-stl-tutorial/
4. http://www.learncpp.com
5. https://www.geeksforgeeks.org/c-magicians-stl-algorithms/ (stl algorithm)
6. http://www.tutorialspoint.com/cplusplus/
7. Object-oriented Programming with C++ by E Balagurusamy