CSCI 2270 - Data Structures and Algorithms
Instructor: Hoenigman / Jacobson
Recitation 11

In this recitation, you will be learning a basic understanding of templates and implementation of the Standard Template Library.

Basic Understanding of Templates

In order to understand the Standard Template Library, we want to introduce you to templates. A template is a C++ feature for generic programming and will permit functions and/or classes to operate with generic types; meaning programmers can write code in a way that does not depend on a specific data type. This allows functionality of different data types without re-writing an entire section of code for each type, thus saving time. This is mostly done in more advanced programming where programmers write their own libraries.

Up to this point, we have seen functions or classes which operate only for specific types. For example, when we wrote a function that takes in arguments of integers, then it will only operate on integer data types and return an integer type. Templates, however, allow us to define a function that can operate on any data type. Therefore, that function can be called multiple times using different data types and left to the compiler to understand.

Template Syntax

Templates are recognized by the use of the less/greater than brackets, (<) and (>). The common format of declaring a template will be:

template < *typename T* >

Where T is a placeholder data type name, which will be specified later. Also, the keyword *typename* can can be replaced with the keyword *class* and will have the same meaning or behavior. It's just the way to declare a template. This would be used when you are making your own templates.

Example

In this example, TYPE can be of any type and the code can be reused regardless of that data type. It will avoid any unnecessary repetition of code for the same task.

```
template < class TYPE >
TYPE add2numbers(TYPE num1, TYPE num2){
        TYPE result = num1 + num2;
        return result;
}
```

If we wanted to send in ints, or floats, the same code would be called and would be handled by the compiler. How does the compiler know the TYPE of result? When it tries to compile this function, the compiler will not care of the type, but rather check syntax and report any errors. Though templates are more involved, we want to move to the Standard Template Library.

Standard Template Library (STL)

The STL is a C++ library that is mostly made up of built-in template classes, which provide a set of classes and functions to implement algorithms and data structures like queues, stacks, or vectors. The STL is mostly used for reusability, efficiency, flexibility, or maintainability of code. We use it so that we don't have to manually write out the queue, or stack, or dynamic arrays.

Previously, we have used C++ standard libraries like I/O or strings, which allowed us to include the library and use an extensive set of properties that we didn't have to place without our own code. This is the same idea for the STL. It has components of containers, algorithms, and iterators that allow us to use each of the STL properties. The STL includes several classes, but we will only focus on stack, queue, and vectors for examples. Also, instead of making your own template from the above examples, we are going to look at vectors using the STL, a standardized template.

Container classes

The most common STL library functionality are the container classes. A *container* is an object that can represent a group of elements of a certain type. A *container class* is a class that will hold and organize multiple instances of the same type. C++ prefers these types to remain the same. The purpose of the STL container class is to contain other objects, which include classes like vectors. They are templates that can contain any type when instantiated.

For example, to declare them you will:

```
#include <queue>              // include the appropriate headers.
                              // setup and use the instance
std::queue <int> myqueue;     // where this will be a queue of integers named myqueue.
myqueue.functionCall();       // functionCall will be any function within the queue class
```

Another example using vectors[1]:

```
#include <vector>
using namespace std;

vector<int> v(3);             // Declare a vector of 3 elements.
```

[1] https://www.sgi.com/tech/stl/stl_introduction.html and
http://www.cs.northwestern.edu/~riesbeck/programming/c++/stl-iterators.html

```
v[0] = 7;
v[1] = v[0] + 3;
v[2] = v[0] + v[1];                    // where v[0] == 7, v[1] == 10, v[2] == 17
```

Algorithms

There are several algorithms within the STL that allow us to manipulate the data that is stored within our containers. They are a predefined set of tools or algorithms that we can pull from and use on any of the container classes. This is different from the methods that can be used on the individual classes. For example, vectors and queues have different methods within them that can be used. Queue has a method that can be called known as .push() whereas vectors have a method that can be called known as .push_back(). These methods aren't the same and you can't access each of them across the class container, but you can access .reverse() for both of them. Reverse is known as a global function, not a member function and is decoupled from the STL container classes. Therefore, it can be used to reverse elements in vectors, or in other container classes.

For example:

```
reverse(v.begin(), v.end());          // where v[0] == 17, v[1] == 10, and v[2] == 7
```

In this example, begin and end are part of the vector methods that can be used when a vector is instantiated. When using the reverse algorithm, you can reverse the order of the elements inside the container class (vector).

Iterators

Lastly, we have iterators which are pointer-like objects that are generated by the STL container member functions, such as .begin or .end. You can think of them in the same way of using pointers. They can be incremented with (++), dereferenced with (*), or can be compared against another iterator with (!=) or some operator.

Iterators work in the same manner as a *for loop*. For Example:

```
int i;
for(i = begin, i < end, i++)          // where i is known as an index or "iterator"

vector<int>::iterator iter;           // an iterator for a vector of ints named iter
for (iter = v.begin(); iter != v.end(); iter++)
    cout << (*iter) << endl;              // points to the the address position of
vector v
```

In class work

For the in class work, you will be writing code to work with queues using the STL. In order to complete this exercise, you will need the following:

- Add appropriate include for queue
- Create/declare an instance of a queue using the STL
- Print the size of the empty queue (*.size()* to print/find the size of the queue)
- Insert [3, 6, 5] into the queue (*.push()* to add the elements onto the queue)
- Print the size after the elements are inserted
- Check if queue empty and empty the queue if not (*.empty()* to check if the queue is already empty and *.pop()* to remove the elements from the queue if not empty)
- Print the size after the elements are deleted

Recitation 11 Programming Assignment

Download the recitiaton11.cpp file to download from Moodle. There is a basic coding structure provided that you will build from. Your goal will be to add code where there is a TODO listed. This assignment will focus on the STL vector class, where you will push, print, reverse, and remove elements from the vector.

You will have until Sunday at 5pm to complete.