CS490: Problem Solving in Computer Science

Lecture 4: C++ Standard Template Library

Dustin Tseng
Mike Li

Wednesday January 4, 2006

Outline

- C++ Templates
- STL Data Structures
- STL Algorithms

- C++ Templates
- STL Data Structures
- STL Algorithms

Introduction

- We use template to allow C++ functions to have type parameters.
- In Java 1.5, there is generics, which is similar to this concept.
- Our focus will be on how to use the C++ STL.

Example

The following `min` function is only defined for `int`:

```
int min( int a, int b ) {
    return a < b ? a : b;
}
```

but we can make it general for any type by using template:

```
template<class C>
int min( C a, C b ) {
    return a < b ? a : b;
}
```

What will happen?

```
min("Xander", "Willow");

min(15, "Giles");
```

- C++ Templates
- STL Data Structures
- STL Algorithms

Overview

The STL provides a large number of data structures. Here are some of the most commonly used ones:

vector : contain contiguous elements stored as an array

queue : FIFO data structure

stack : FILO data structure

list : sequences of elements stored in a linked list

set : sorted set of unique objects

map : sorted associative containers that contain unique key/value pairs

Reference: `http://www.cppreference.com/`

Vectors

Vectors work much like a normal array.

- To create an array of intergers with maximum capacity `n`:

```
vector<int> V(n);
```

- to access an element:

`V.at(i)` or `V[i]`

- to append an element:

```
V.puch_back(c);
```

- to insert an element:

```
V.insert(loc, c);
```

- to acess the number of elements:

```
V.size();
```

- what does `V.max_size()` return?
- what would happen when we `push_back` when `V` has already reached its `max_size`?

Iterators

When inserting an element into a vector, we used `V.insert(loc, c)`, here `loc` is a `vector<int>::iterator` Iterators are used to access members of the container classes. We can view them as pointers. e.g. to print the members of a vector:

```
vector<int>::iterator it;
for(it = V.begin(); it != V.end(); it++)
    cout << *it;
```

To print the members in the reverse order, we have to use reverse iterators:

```
vector<int>::reverse_iterator it;
for(it = V.rbegin(); it != V.rend(); it++)
    cout << *it;
```

A GNU g++ trick: we can use `typeof` to simplify the declaration of interators. e.g. we could use

```
for(typeof(V.rbegin()) it = V.rbegin(); it != V.rend(); it++)
```

Queues, Stacks and Deques

- Queues: First-In-First-Out
- Stacks: First-In-Last-Out
- Deques: Double-Ended Queues

```
#include <stack>
#include <queue>
// either one will provide deque

queue< int > Q;              // Construct an empty queue
for ( int i = 0; i < 3; i++ )
    Q.push( i );             // Pushes i to the end of the queue

// Q is now { 0, 1, 2 }
int sz = Q.size();           // Size of queue is 3
while( !Q.empty() ) {        // Print until Q is empty
    int element = Q.front(); // Retrieve the front of the queue, top() for stack
    Q.pop();                 // REMEMBER to remove the element!
    cout << element << endl; // Prints queue line by line
}
```

Lists

Lists in STL are doubly-linked lists. Some list operations are similar to deque:

- `front()`, `back()`
- `pop_front()`, `pop_back()`
- `push_front()`, `push_back()`

List does not have `at()`, but it supports the following:

- `sort()` : order a list
- `splice()` : join two lists
- `merge()` : join and order two lists

Sets

STL sets is a sorted collection of elements stored as a balanced binary search tree. In order to use `set<C>`, the < operator must be defined for `C`. e.g. < operator is defined in `string`

```
set< string > ss;
ss.insert( "Sheridan" );
ss.insert( "Delenn" );
ss.insert( "Lennier" );
for( set< string >::iterator j = ss.begin(); j != ss.end(); ++j )
    cout << " " << *j;
```

What is the complexity of `insert()`?
What will be the output?

More Sets

An easy way to implement tree sort:

```
vector< int > v = {... };
set< int > s( v.begin(), v.end() );
vector< int > w( s.begin(), s.end() );
```

What is the complexity?

We use `count()` to check whether an element exists, and use `erase()` to erase elements. A range of element can be erased by passing in two iterators.

```
set< int > s;
for( int i = 1900; i <= 3000; i += 4 ) s.insert( i );
for( int i = 1900; i <= 3000; i += 100 ) if( i % 400 != 0 ) {
    if( s.count( i ) == 1 ) s.erase( i );
}
```

Maps

Like STL sets, STL maps also store elements in a sorted fashion. However, the elements in a map are key-value pairs. We may view a map as a dictionary, where the paired elements are sorted by the keys. Since map is implemented using balanced binary search trees, insertion, deletion and update operations require logarithmic time in the size of the map.

e.g. mapping days of a week

```
vector< string > i2s( 7 );
i2s[0] = "Sunday"; i2s[1] = "Monday"; i2s[2] = "Tuesday"; i2s[3] = "Wednesday";
i2s[4] = "Thursday"; i2s[5] = "Friday"; i2s[6] = "Saturday";
map< string, int > s2i;
for( int i = 0; i < 7; i++ ) s2i[i2s[i]] = i;
```

More Maps

e.g. word frequency

```
string word;
map< string, int > freq;
while( cin >> word ) freq[word]++;
```

to print the key-value pairs in a map:

```
for( map< string, int >::iterator i = freq.begin(); i != freq.end(); ++i)
    cout << (*i).first << ": " << (*i).second << endl;
```

`first()` and `second()` are `pair` operations. This is because STL `map` actually uses a sub data structure `pair`. Even when we just want to use pairs, we have to `#include<map>`.

Maps Again

We should always use `count(key)` to check whether a given key exists in a map. This is because the fact that if we call:

```
map<string, int> freq;
cout << freq.size() << endl;
if(freq["something"] > 0)
  cout << "hah!" << endl;
cout << freq.size() << endl;
```

although we won't be laughed at, since freq["something"] will indeed be 0, we have created a possiblly unwanted element in the map, by merely calling it.

The output from the code above would be:

```
0
1
```

Comparators

Sometimes we may want to modify the way we compare things when we store them in a map or a set. For example, by default, the < operatior on strings performs lexicographic comparison. If we want to sort strings first by length, here is what we can do:

```
struct myLessThan {
    bool operator()( const string &s, const string &t ) const {
        if( s.size() != t.size() ) return s.size() < t.size();
        return s < t;
    }
};

int main() {
    set< string, myLessThan > coll;
    coll.insert( "cat" );
    coll.insert( "copper" );
    coll.insert( "cow" );
    coll.insert( "catch" );
    for( set< string, myLessThan >::iterator i = coll.begin(); i != coll.end(); ++i )
        cout << " " << *i;
    return 0;
}
```

What will the output be?

- C++ Templates
- STL Data Structures
- STL Algorithms

Sorting

The algorithm library in C++ provides serveral common and useful algorithms. To use them, have `#include <algorithm>`

Sorting is a very common task, and can be done very easily with the help of `sort()`. To use `sort()`, we pass in two iterators that define the range of the operation.

```
vector<int> v;
v.push_back( 3 );
v.push_back( 1 );
v.push_back( 2 );
sort( v.begin(), v.end() ); // sort v in increasing order
```

Customized Sort

We may customize `sort()` by once again, defining our own comparator:

```
struct object {
int weight, cost;
};

struct ComparisonFunctor {
bool operator() ( const object &a, const object &b ) const {
if ( a.cost != b.cost ) return a.cost < b.cost;
else return a.weight > b.weight;
}
};

vector<object> vo;
// ... insert some objects to the vector
// A. sort the vector first by lowest cost, then by greatest weight
sort( vo.begin(), vo.end(), ComparisonFunctor() );

// B. we can also define a set using the functor.
// the following copies the vector to the set, which will be kept in
// sorted order (Binary Search Tree)
set< object, ComparisonFunctor > myset( vo.begin(), vo.end() );
```

Customized Sort

Although a function object, or functor, gives us the advantage of re-using it in sets or maps, often we may find it easier just to use a function directly:

```
bool compare( const object &a, const object &b ) {
    if ( a.cost != b.cost ) return a.cost < b.cost;
    else return a.weight > b.weight;
}
```

and to use it:

```
sort( vo.begin(), vo.end(), compare );
```

Notice how this time there is no brackets.

Permutation

- `next_perputation` is another very useful function comes with algorithm. We often need it in brute force methods. Once again, this function takes in two iterators and operates on a range of values.
- For a finite set $S = 1, 2, ..., n$, the permutation would be $n!$. If we allow duplicates, then the formula becomes $\frac{n!}{c_1!c_2!...c_m!}$, where c_i represents the number of duplicates for ith element.
- e.g. For a set of 3 elements, there are $3! = 6$ different permutations: (1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2) and (3,2,1).
- e.g. There are 10 permutations of (1,1,2,2,2). They are (1,1,2,2,2), (1,2,1,2,2), (1,2,2,1,2), (1,2,2,2,1), (2,1,1,2,2), (2,1,2,1,2), (2,1,2,2,1), (2,2,1,1,2), (2,2,1,2,1) and (2,2,2,1,1).

Permutation

- First notice that there is always a way to sort the permutations lexicographically.
- `next_perputation` actually takes a range of elements and change it to the lexicographically next permutation.
- What will happen if we start with an arrangement that is already the lexicographically largest permutation?
- To ensure we go though all permutations, we usually first sort the range, and combo `next_permutation()` with a `do-while` loop.

```
vector< int > v;
// ...fill in v with some integers
sort( v.begin(), v.end() );
do {
    for( int i = 0; i < ( int )v.size(); i++ ) cout << " " << v[i];
    cout << endl;
} while( next_permutation( v.begin(), v.end() ) );
```

What Else?

Someone demonstrates Halloween Again!