More on STL Containers

STL Containers

- deque (pronounced "deck", short for double ended queue)
- list
- set / multiset
- map / multimap

deque

- deque is a sequence container
- This means the elements are stored in the same order that they were inserted (i.e., they do not get re-ordered when inserted).
- The memory allocated within a deque for storage is not contiguous like it is for a vector.
- The memory is allocated in chunks or pages.
- It allows random access, like a vector(i.e., operator [] is overloaded)
- Search in a deque is linear, i.e., complexity is O(n).

deque

- deque is suitable for insertion and removal at the beginning (unlike vector).
- Removal or insertion of elements at beginning or end is done in constant time (unlike vector, which provides constant time only for removal and insertion at end).
- Removal or insertion of elements in the middle is linear complexity (minimum of number of elements from this point to beginning or the end).
- As a result, deque has the following two methods in its interface:
  - push_front // insert element at front of queue
  - pop_front // remove element from front of queue
- deque does not provide **capacity()** or **reserve()** methods.

deque

- Since deque does not allocate memory contiguously, there is one level of indirection for accessing elements.
  - As a result, element access and traversal is a little bit slower than vector.

- The interface for a vector and deque is very similar except for the methods pointed out above.

- As long as you do not use those methods, you could switch between a vector and a deque if you wanted to do performance runs.

- You can use a deque when you have insertions or deletions at the front of the queue.

List

- list is a sequence container
- This means the elements are stored in the same order that they were inserted (i.e., they do not get re-ordered when inserted in a list).
- In STL, list is implemented as a doubly linked list.
- It does NOT allow random access, unlike a vector.
- Traversal in a list is linear, i.e., complexity is O(n).
- Since random access is not supported on a list, the operator [] is not overloaded on the list class (unlike vector).

List

- Inserting or deleting anywhere within the list is fast, since elements need not be shifted.

- Compare this with a vector, where :
  - If you had to insert or delete the first element, you would need to shift all the rest of the elements, and so the complexity would be O(n).
  - If you had to insert at end or delete at end, the complexity would be O(1).
  - If you had to insert in the middle or delete in the middle, the complexity would be O(n/2), which really means O(n).

List

- Have to use iterators to traverse the list
  - can use iterators in vector also, and in fact, for any STL container.

- Member method **front()** returns the first element.
- Member method **back()** returns the last element.
- **size()** returns the number of elements in the list.
- No **capacity()** or **reserve()** methods as they don't make sense on a list.
- The elements in a list have to have the following characteristics :
  - Assignable
  - Copyable or movable

Simple Example

A simple example showing list usage:

```
#include <list>
using std::list;

int main()
{
        list <int> li;   // a list of integers

        for (int ii = 0; ii < 20; ++ ii)
                li.push_back( ii );          // or can call li.push_front( ii )


        // Now we iterate over the list
        list <int>::const_iterator citer = li.begin();


        for ( ; citer != li.end(); ++ citer )
                cout << "Integer at position " << ii << " is " << *citer << endl;
}
```

Swap

Example showing swapping contents of two lists:

```
list <double> l1;
list <double> l2;

swap(l1, l2)
```

Or

l1.swap(l2) will swap the two lists.

- Time complexity for executing swap is constant, i.e., it does not depend on the size of the lists.

Splice

- Lists have the capability to remove one or more elements at any position and insert these into any position in another (or same) list.
- No copying of elements is done, as it is a matter of redirecting the pointers.

```
list <double> l1,  l2;
// Move elements of l2 into l1, before position l1Pos
l1.splice( l1Pos, l2 );

// move element of l2 in position l2Pos, into l1, before position l1Pos
l1.splice( l1Pos, l2, l2Pos);
```

Splice

// move element of l2 starting from l2PosBeg and until one before l2PosEnd,
// into l1, before position l1Pos.

// This range is known as the half open range, denoted as [l2PosBeg, l2PosEnd).
// This means starting from (and including) l2PosBeg, up to l2PosEnd, but **not** including l2PosEnd.

l1.splice(l1Pos, l2, l2PosBeg, l2PosEnd);

set/multiset

- set and multiset are **associative** containers.
- When you insert elements into a set or multiset, they are stored in a sorted manner.
- These are efficient for searching.
- Search complexity for a vector or a deque or a list is linear, but for sets and multisets it is O(log N), were N is the number of elements.
- The standard does not specify an implementation, but these are usually implemented as balanced binary trees.

set/multiset

- The difference between a set and a multiset is that the former (set) allows only one copy of a given value, whereas the latter allows multiple copies.

- The elements put in a set should be:
  - Assignable
  - Copyable or movable
  - Comparable (so that they can be sorted).

- Once an element is put into a set, its value should not be changed in place.

  If you need to change a value, you have to put it back into the set.

set/multiset

- You can specify your sorting criteria for a set or multiset.
- If you do not, then the default criteria is less.
  - This will sort based on comparing with the operator <.
- If I say the following, I will get the default criteria of less.
  - set< int > myIntSet; // this defaults to less<int>, which uses <
- If I want to specify a different criteria, I can do the following:
  - set< int, std::greater<int> > myIntSet2; // this uses >

set/multiset

I can of course use typedef to simplify:

```
typedef   std::set< int >                          TMyIntSet1;
typedef   std::set< int, std::greater<int> >       TMyIntSet2;
```

And now my declarations don't look as bad ☺

```
TMyIntSet1      set1;
TMyIntSet2      set2;
```

- Note that even though these are both sets of integers, their types are different since the comparator is different.

map/multimap

- map and multiMap are also associative containers.
- The difference between map and set is that map is a key,value pair, whereas in the set, you only have a value.
- Sorting (and searching) is done based on the key, not the object (unlike sets).
- Set is like a map, except that it does not contain the (key, object) pair.
  - In this, the objects are sorted based on their own values.
- The standard does not specify an implementation, but these are usually implemented as balanced binary trees. .
- multimap is a map that allows duplicate keys.
  - In a map, there cannot be more than one entry for a given key.

map/multimap

- The keys put in a map or multimap have to have the following characteristics :
  - Assignable
  - Copyable
  - Comparable (so that they can be sorted).

- The values put in a map or multimap have to have the following characteristics :
  - Assignable
  - Copyable or movable

- The subscript operator [] is overloaded for map and multimap class, so that you can use these as associative containers, i.e., you can access the values by specifying the key as an index, in other words, specifying the key as an argument to the subscript operator.

map/multimap

- Similar to set, the default sorting criteria is less.
  - This will sort based on comparing with the operator <

- map< int, int > myIntMap; // this defaults to less<int>, which uses <
- map< int, MyClass > myMap1; // maps an integer to MyClass object
- map< int, MyClass * > myMap2; // maps an integer to MyClass pointer

- For myMap1 above, when I retrieve the value from the map, I will get a reference to the object.
- Example:

  - MyClass & value1 = myMap1[4];
  - // value1 is a reference to the object stored in map

map/multimap

- I can also do the following, but keep in mind that this is creating an extra copy of the MyClass object that was retrieved from the map.
  - MyClass value2 = myMap1[4];
  - //value2 is a copy of object stored in map

- It is very important to note the following behavior of maps and multimaps:
- When you use the subscript operator [], it will do one of the two things below:

1. If the entry exists in the map, a reference to the value is returned.

2. If an entry does NOT exist in the map, it will CREATE one.

map/multimap

- This tells you the following:

```
map< int, MyClass * > myMap2;
MyClass * value = myMap2[10];
if ( value != NULL )
        cout << "value found" << endl;
else
        cout << "value not found" << endl;
```

- When you execute this, if entry with key 10 was not there in the map, it is now created based on rule #2.
- It will have a default value, which in this case will be a NULL pointer.

- Takeaway: operator [] can have a side effect if used "carelessly".

map/multimap

- The right way to check this is to call the find method which returns an iterator the element if it is present, or the end iterator if not present (See code example in code file).

- Using it with the find method does NOT have a side effect.

map/multimap

- Inserting elements into the map looks like the following, which inserts a (key, value) pair of (10, 100) :
  - myIntMap.insert(std::make_pair(10, 100));

- Or in myMap1, we insert an integer key and a MyClass object:

```
MyClass m1( some args );
myMap1.insert (std::make_pair(m1.GetID(),  m1) );
// assuming MyClass has a GetID method.
```

- Or in myMap2, we insert an integer key and a MyClass pointer :

```
MyClass * m2Ptr = new MyClass( some args );
myMap2.insert (std::make_pair (m2Ptr->GetID(),  m2Ptr) );
```

map/multimap

- make_pair is an STL function that simply makes a pair and returns that pair, and in this case we get back a (key, value) pair and insert that into the map (See code example in code file for usage).

Container categories

- Associative containers (for fast searches)
  - set, multiset
  - map, multimap

- Unique associative containers (do not allow duplicates)
  - set
  - map

- Multiple associative containers (allow duplicates)
  - multi set
  - multi map

- Simple associative container (elements also act as keys)
  - set
  - multiset

Container categories

- Pair associate containers
  - map, multimap
  - hash_map, hash_multimap
- Sorted associate containers
  - map, multimap
  - set, multiset
- Hashed associative containers
  - hash_map, hash_set
- Multiple Hashed associative containers
  - hash_multimap, hash_multiset