STL Containers, Part II

Introduction

Last week we talked about `vector` and `deque`, the STL's two managed array classes. However, the STL offers many other containers, two of which, `set` and `map`, we will cover in this handout. Because `set` and `map` are associative containers, they require a decent understanding of iterators and iterator syntax. At the end, we'll quickly talk about two special container classes, the `multimap` and `multiset`, which have no counterparts in the CS106 ADT library.

Using `typedef` Liberally

Now that you're using STL iterators more frequently, you might find it useful to liberally sprinkle the `typedef` statement to create shorthand for your types. For example, if you have a `set<string>`, the return type of its `insert` function will be `pair<set<string>::iterator, bool>`. Do you really want to have to type that out every single time?

While it's by no means a requirement, you should strongly consider `typedef`ing iterator types if you plan on using them frequently. For example, if you `typedef set<string>::iterator` as `SetStrItr`, then the syntax for the return type of `insert` reduces to `pair<SetStrItr, bool>`, which is still mouthful but is considerably better than the first.

`set`

The first container we'll cover today is the `set`. Like the CS106 `Set`, `set` is an ordered container representing an abstract collection of objects. You can test a `set` for inclusion, and can also insert and remove elements. However, unlike the CS106 `Set`, the STL `set` does not have intrinsic support for union, intersection, and difference, though next week we'll see how you can mimic the functionality using STL algorithms.

Like the CS106 `Set`, when storing non-primitive types, the `set` needs extra information about how to compare those objects. Unfortunately, the syntax to provide a comparison callback for the `set` either uses complicated, error-prone syntax or requires an understanding of operator overloading, a language feature we haven't covered yet. Thus, for now, you should avoid storing custom data types in `set`s.

The most basic operation you can perform on a `set` is insertion using the `insert` function. Unlike the `deque` and `vector` `insert` functions, you do not need to specify a location for the new element because the `set` automatically orders its elements. Here is some sample code using `insert`:

```
set<int> mySet;
mySet.insert(137);  // Now contains: 137
mySet.insert(42);   // Now contains: 42 137 (in that order)
mySet.insert(137);  // Now contains: 42 137 (note no duplicates)
```

To check whether the `set` contains an element, you can use the `find` function. `find` searches the `set` for an element and, if found, returns an iterator to it. Otherwise, `find` returns the value of the container's `end` function as a sentinel indicating the element wasn't found. For example, given the `set` initialized above:

```
if(mySet.find(137) != mySet.end())
    cout << "The set contains 137." << endl; // This is printed.

if(mySet.find(0) == mySet.end())
    cout << "The set doesn't contain 0." << endl; // Also printed.
```

Instead of `find`, you can also use the `count` function, which returns 1 if the element is contained in the set and 0 otherwise. Using C++'s automatic conversion of positive values into `true` and zero values to `false`, you can actually write much cleaner code using `count`. For example:

```
if(mySet.count(137))
    cout << "137 is in the set." << endl;  // Printed
if(!mySet.count(500))
    cout << "500 is not in the set." << endl; // Printed
```

If `count` is simpler than `find`, why use `find`? The reason is that sometimes you'll want an iterator to an element of a `set` for reasons other than determining membership. You might want to erase the value, for example, or perhaps you're trying to obtain iterators to a range of values for use with an STL algorithm. Thus, while `count` is quite useful, `find` is much more common in practice.

You can remove elements from a `set` using `erase`. `erase` has several versions, one of which removes the value pointed at by the specified iterator, and another that removes the specified value from the `set` if it exists. For example:

```
mySet.erase(137); // Removes 137, if it exists.
set<int>::iterator itr = mySet.find(42);
if(itr != mySet.end())
    mySet.erase(itr);
```

Sometimes it can be useful to search the `set` for all elements within a certain range. For example, suppose we wanted to obtain iterators to all values in the `set` in the range [10, 100]. Using only the above functions, we'd have to iterate manually over the entire `set` looking for values that matched the criteria. However, `set` exports two useful functions called `lower_bound` and `upper_bound` that can greatly simplify this task. `lower_bound` returns an iterator to the first element in the `set` greater than or equal to the specified value, while `upper_bound` returns an iterator to the first element in the `set` that isn't less than or equal to the specified element. Using `lower_bound` to get an iterator to the first element in the range and iterating until we reach the value returned by `upper_bound`, we can loop over all elements in a the range. For example, the following loop iterates over all elements in the set in the range [10, 100]:

```
set<int>::iterator stop = mySet.upper_bound(100);
for(set<int>::iterator itr = mySet.lower_bound(10); itr != stop; ++itr)
     // ... perform tasks...
```

A word of caution: you should not modify items pointed to by `set` iterators. Since `set`s are internally stored in sorted order, if you modify a value in a `set` in-place, you risk ruining the internal

ordering. With the elements no longer in sorted order, the `set`'s searching routines may fail, leading to difficult-to-track bugs. If you want to apply a transformation to a `set`'s elements, unless you're convinced the transformation will leave the elements in forward-sorted order, you should strongly consider creating a second `set` and storing the result there.

The following table lists some of the most important `set` functions, though there are more. The entries in this table are also applicable for the `map`.

Again, we haven't covered `const` yet, so for now it's safe to ignore it. Also, we haven't covered `const_iterators`, but for now you can just treat them as iterators that can't write any values.

Constructor: `set<T>()`	`set<int> mySet;` Constructs an empty `set`.
Constructor: `set<T>(const set<T> &other)`	`set<int> myOtherSet = mySet;` Constructs a `set` that's a copy of another `set`.
Constructor: `set<T>(InputIterator start, InputIterator stop)`	`set<int> mySet(myVec.begin(), myVec.end());` Constructs a `set` containing copies of the elements in the range [`start`, `stop`). Any duplicates are discarded, and the elements are sorted. Note that this function accepts iterators from any source.
`size_type size() const`	`int numEntries = mySet.size();` Returns the number of elements contained in the `set`.
`bool empty() const`	`if(mySet.empty()) { ... }` Returns whether the `set` is empty.
`void clear()`	`mySet.clear();` Removes all elements from the `set`.
`iterator begin()` `const_iterator begin() const`	`set<int>::iterator itr = mySet.begin();` Returns an iterator to the start of the `set`. Be careful when modifying elements in-place.
`iterator end()` `const_iterator end()`	`while(itr != mySet.end()) { ... }` Returns an iterator to the element one past the end of the final element of the `set`.

`pair<iterator, bool>`
`insert(const T& value)`
`void insert(InputIterator begin,`
`InputIterator end)`
	`mySet.insert(4);`
	`mySet.insert(myVec.begin(), myVec.end());`
	
	The first version inserts the specified value into the `set`. The return type is a `pair` containing an iterator to the element and a `bool` indicating whether the element was inserted successfully (`true`) or if it already existed (`false`). The second version inserts the specified range of elements into the `set`, ignoring duplicates.

`set` functions, contd.

`iterator find(const T& element)`	`if(mySet.find(0) == mySet.end()) { ... }` Returns an iterator to the specified element if it exists, and `end` otherwise.
`size_type count(const T& item) const`	`if(mySet.count(0)) { ... }` Returns 1 if the specified element is contained in the `set`, and 0 otherwise.

`size_type erase(const T& element)`
`void erase(iterator itr);`
`void erase(iterator start,`
`iterator stop);`
	`if(mySet.erase(0)) {...} // 0 was erased`
	`mySet.erase(mySet.begin());`
	`mySet.erase(mySet.begin(), mySet.end());`
	
	Removes an element from the `set`. In the first version, the specified element is removed if found, and the function returns 1 if the element was removed and 0 if it wasn't in the `set`. The second version removes the element pointed to by `itr`. The final version erases elements in the range [`start`, `stop`).

`iterator lower_bound(const T& value)`	`itr = mySet.lower_bound(5);` Returns an iterator to the first element that is greater than or equal to the specified value. This function is useful for obtaining iterators to a range of elements, especially in conjunction with `upper_bound`.
`iterator upper_bound(const T& value)`	`itr = mySet.upper_bound(100);` Returns an iterator to the first element that is greater than the specified value. Because this element must be strictly greater than the specified value, you can iterate over a range until the iterator is equal to `upper_bound` to obtain all elements less than the parameter.

`pair`

Before starting `map`, let us first quickly discuss `pair`, a template `struct` that simply stores two elements of mixed types. `pair` accepts two template arguments and is declared as `pair<TypeOne, TypeTwo>`. `pair` has two fields, named `first` and `second`, which store the values of the two elements of the pair: `first` is a variable of type `TypeOne`, `second` of type `TypeTwo`.

You can create a pair explicitly using this syntax:

```
pair<int, string> myPair;
myPair.first = myInteger;
myPair.second = myString;
```

Alternatively, you can initialize the `pair`'s members in the constructor as follows:

```
pair<int, string> myPair(myInteger, myString);
```

In some instances, you will need to create a `pair` on-the-fly to pass as a parameter (especially to the `map`'s `insert`). You can therefore use the `make_pair` function as follows:

```
pair<int, string> myPair = make_pair(137, string("string!"));
```

Note that when passing a string literal to `make_pair` you must manually typecast the string from its normal C string form into an actual C++ `string`. `make_pair` will automatically return a `pair` of the types of its arguments, and unless you manually force C strings to become C++ `string`s, you'll get back a `pair<int, char *>` instead of a `pair<int, string>`. This is an easy mistake to make, but can generate a slew of awful compiler errors. Of course, if you truly want a `pair<int, char *>`, avoid the typecast.

map

The `map` is one of the STL's most useful containers. Unlike the CS106 `Map`, the STL `map` lets you choose what key type you want to use. That is, the STL `map` can map `string`s to `int`s, `int`s to `string`s, or even `vector<int>`s to `deque<double>`s. Also unlike the CS106 `Map`, the STL `map` stores its elements in sorted order. That is, if you iterate over the keys in a `map<string, int>`, you'll get all of the `string` keys back in alphabetical order.

If you want to use nonstandard types as keys in a `map` – for example, your own custom `struct`s – you will need to provide a comparison function. Again, like `set`, providing this function is quite difficult and we will not discuss how until we cover operator overloading.

To declare a `map`, use the following syntax:

```
map<KeyType, ValueType> myMap;
```

For example, to create a `map` similar to the CS106 `Map<int>`, you'd write `map<string, int>`, a mapping from `string`s to `int`s.

Unfortunately, while the STL `map` is quite powerful, it is a syntax nightmare because its functions and iterators need to accept, return, and manipulate two values. Thus the `map`'s iterators act as pointers to `pair<const KeyType, ValueType>`s, a pair of an immutable key and a mutable value. For example, to iterate over a `map<string, int>` and print out all its key/value pairs, you can use the following loop:

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

Note that the key is `itr->first` and the value is `itr->second`. While you cannot change the value of the key during iteration,* you can modify its associated value.

As with the `set`, you can use `find` to return an iterator to an element in a `map` given a key. For example, to check to see if "STL" is a key in a `map<string, int>`, you can use this syntax:
`if(myMap.find("STL") != myMap.end()) { ... }`

---

* As with the `set`, changing the values of the keys in a `map` could destroy the internal ordering and render the `map` non-functional. However, unlike the `set`, if you try to modify a key with an iterator, you'll get a compile-time error.

Alternatively, you can use the `count` function as you would with a `set`.

Because the `map` stores everything as `pair`s, inserting a new value into a map is a syntax headache. You must manually create the pair to insert using the `make_pair` function discussed earlier. Thus, if you wanted to insert the key/value pair ("STL", 137) into a `map<string, int>`, you must use this syntax:

```
myMap.insert(make_pair(string("STL"), 137));
```

Unlike the CS106 `Map`, if you try to overwrite the value of an existing item with the `insert` function, the STL `map` won't modify the value. Thus, if you write

```
myMap.insert(make_pair(string("STL"), 137)); // Inserted
myMap.insert(make_pair(string("STL"), 42));  // Whoops!  Not overwritten.
```

The value associated with the key "STL" will still be the value 137. To check whether your insertion actually created a new value, the `map`'s `insert` function returns a `pair<iterator, bool>` that contains information about the result of the operation. The first element of the `pair` is an iterator to the key/value pair you just inserted, and the second element is a `bool` indicating whether the `insert` operation set the value appropriately. If this value is `false`, the value stored in the `map` was not updated. Thus, you can write code like this to insert a key/value pair and manually update the value if the key already existed.:

```
// Try to insert normally.
pair<map<string, int>::iterator, bool> result;
result = myMap.insert(make_pair(string("STL"), 137));

// If insertion failed, manually set the value.
if(!result.second)
    result.first->second = 137;
```

In the last line, the expression `result.first->second` is the value of the existing entry, since `result.first` yields an iterator pointing to the entry, so `result.first->second` is the value field of the iterator to the entry. As you can see, the `pair` can make for tricky, unintuitive code.

There is an alternate syntax you can use for insertion involving the bracket operator. As with the CS106 `map`, you can access individual elements by writing `myMap[key]`. If the element doesn't already exist, it will be created. Furthermore, if the key already exists, the value will be updated. For example, you can use the bracket operator to insert a new element into a `map<string, int>` as follows:

```
myMap["STL"] = 137;
```

You can also use the bracket operator to read a value from the `map`. As with insertion, if the element doesn't exist, it will be created. For example:

```
int myValue = myMap["STL"];
```

If the bracket notation is so much more convenient, why even bother with `insert`? As with everything in the STL, the reason is efficiency. Suppose you have a

`map<string, vector<int> >`. If you insert a key/value pair using `myMap.insert(make_pair(string("STL"), myVector))` the newly created `vector<int>` will be initialized to the contents of `myVector`. With `myMap["STL"] = myVector`, the bracket operation will first create an empty `vector<int>`, then assign `myVector` on top of it. This requires more allocations and deallocations and consequently is much slower.

Again, as an STL programmer, you are working with professional libraries. If you want to go for raw speed, yes, you will have to deal with more complicated syntax, but if you want to just get the code up and running, then the STL won't have any reservations.

`multimap` and `multiset`

The STL provides two special "multicontainer" classes, `multimap` and `multiset`, that act as `map`s and `set`s except that the values and keys they store are not necessarily unique. That is, a `multiset` could contain several copies of the same value, while a `multimap` might have duplicate keys associated with different values.

`multimap` and `multiset` have identical syntax to `map` and `set`, except that some of the functions work slightly differently. For example, the `count` function will return the number of copies of an element an a multicontainer, not just a binary zero or one. Also, while `find` will still return an iterator to an element if it exists, the element it points to is not guaranteed to be the only copy of that element in the multicontainer. Finally, the `erase` function will erase *all* copies of the specified key or element, not just the first it encounters.

One function that's quite useful for the multicontainers is the `equal_range` function. `equal_range` returns a `pair<iterator, iterator>` that represents the span of entries equal to the specified value. For example, given a `multimap<string, int>`, you could use the following code to iterate over all entries with key "STL":

```
// Assume we have multimap<string, int> myMultiMap.
// Store the result of the equal_range
pair<multimap<string, int>::iterator, multimap<string, int>::iterator>
   myPair = myMultiMap.equal_range("STL");
// Iterate over it!
for(multimap<string, int>::iterator itr = myPair.first;
itr != myPair.second; ++itr)
   cout << itr->first << ": " << itr->second << endl;
```

As you can see, you might find yourself needing to `typedef` your iterator types a bit more with the multicontainers, but nonetheless they are quite useful.

More to Explore

In this handout we covered `map` and `set`, which combined with `vector` and `deque` are the most commonly-used STL containers. However, there are several others we didn't cover, a few of which might be worth looking into. Here are some topics you might want to read up on:

1. **`list`**: `vector` and `deque` are both sequential containers that mimic built-in arrays. The `list` container, however, models a sequence of elements without indices. `list` supports several nifty operations, such as merging, sorting, and splicing, and has quick insertions at almost any point. If you're planning on using a linked list for an operation, the `list` container is perfect for you.
2. **The Boost Containers**: As mentioned in the `string` handout, the Boost C++ Libraries are a collection of functions and classes developed to augment C++'s native library support. Boost offers several new container classes that might be worth looking into. For example, `multi_array` is a container class that acts as a `Grid` in any number of dimensions. Also, the `unordered_set` and `unordered_map` act as replacements to the `set` and `map` that use hashing instead of tree structures to store their data. If you're interested in exploring these containers, head on over to www.boost.org.

Practice Problems

Some of these are programming exercises, and some of these are just cool things to think about. Play around with them a bit to get some practice with `map` and `set`!

1. Using the time trial code from lecture, time the speed differences of the `map`'s `insert` function versus the bracket syntax in a `map<int, int>`, then repeat the trial for `map<int, string>`. What does this tell you about the time it takes to copy objects?
2. To understand the sheer madness of STL error messages, given a `map<string, int>` called `myMap`, write `myMap.insert(make_pair("STL", 137))` instead of `myMap.insert(make_pair(string("STL"), 137))`. How many errors do you get? What if you forget the call to `make_pair` and instead write `myMap.insert(string("STL"), 137)`?
3. Write a function `NumberDuplicateEntries` that accepts a `map<string, string>` and returns the number of duplicate values in the `map`.
4. As mentioned earlier, you can use a combination of `lower_bound` and `upper_bound` to iterate over elements in the closed interval [min, max]. What combination of these two functions could you use to iterate over the interval [min, max)? What about (min, max] and (min, max)?
5. Write a function `PrintMatchingPrefixes` that accepts a `set<string>` and a `string` containing a prefix and prints out all of the entries of the `set` that begin with that prefix. Your function should only iterate over the entires it finally prints out. You can assume the prefix is nonempty and should treat prefixes case-sensitively. *(Hint: In a `set<string>`, strings are sorted lexicographically, so all strings that start with "abc" will come before all strings that start with "abd.")*