Lists

```
#include <list>
```

The STL **list** class is a doubly-linked list. It allows insertion, deletion and modification of elements at the front, the back and in the middle. Like **deque**, **list** supports **size**, **push_back, push_front, pop_back** and **pop_front**. To work with the internal elements of the list, we need the mechanism of iterators.

Iterators

Vectors, lists, queues, etc. are all examples of collections. In fact, they are all subclasses of a **Container** class in STL. An iterator is like a reference (or pointer) to one of the elements of a collection. At the very least, an iterator supports two operations – advance forward one position (implemented using the ++ operator) and return the element being referenced (implemented using the unary * operator). Here is how we can use an iterator to ouput the elements of a list.

Example 6:

```
list< int > c;
c.push_back( 3 );
c.push_back( 5 );
c.push_front( 6 );
for( list< int >::iterator i = c.begin(); i != c.end(); ++i )
    cout << " " << *i;
```

The first line creates an empty list of integers. The next 3 lines add 3 elements to the list, which now contains 6, 5 and 3, in that order. The **for** loop needs some explanation. "list< int >::iterator" is the class name for an iterator over a list of integers. i is the variable name – this is the iterator. It is initialized to c.begin(). The begin() method of every collection returns an iterator pointing to the first element of the collection. So we have initialized an iterator, i, to reference the first integer in the list. The loop will continue while "i != c.end()". The end() method returns an iterator pointing to the position in the collection that is one after the last element. In other words, when i is equal to c.end(), i has run off the end of the list and is pointing to a non-existent position – the loop should then stop. Finally, ++i increments the iterator at the end of each loop iteration. This moves the iterator one position forward along the list. Inside the loop, we print a space, followed by "*i" – the value that i is referencing. In this case, *i has type int. In general, for a list of type "list< C >", *i would have type C. The code above will print "_6_3_5".

This is the same as using iterators with Java collections, but with a very different syntax. An important difference is the pair of methods, begin() and end(). Think of them as defining a range or an interval [begin, end) that spans the entire collection. The left endpoint is included in the range, and the right endpoint is excluded. This convention is extremely convenient and is used everywhere that ranges are used in the STL. For example, if begin() is equal to end(), it means that the range is empty. You can also pass a range to the constructor of most collections. Here is how we can create a list from a vector.

Example 7:

```
vector< double > v( 10, 7.0 );
list< double > w( v.begin(), v.end() );
vector< double > x( ++w.begin(), w.end() );
```

The first line creates a vector of 10 elements, each of which is 7.0. The second line creates a list containing a copy of the elements in the vector. The third line creates another vector that copies the contents of the list, except for the first element. The vector x will contain 9 elements.

Sets

```
#include <set>
```

The STL **set** is equivalent to Java's java.util.TreeSet. It is a sorted collection of elements, stored as a balanced **binary search tree**. To store elements in a TreeSet, each element must implement the Comparable interface. Similarly, to store elements of type C in a **set**, the < operator must be defined for type C. For example, you can safely declare sets of int, double, char or any othe primitive types because you can use the < operator to compare two ints, two doubles or two chars – the operator is already defined in C++. You can also compare **string**s using the < operator (it is defined in the **string** class); hence, you can declare a set of strings. Here is an example.

Example 8:
```
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;
```

The first line creates an empty set of strings. You need to add "#include <string>" as well as "#include <set>" at the beginning of the program for this to work. The next 3 lines add elements to the set. Because a set is sorted, **push_back** or **push_front** would not make much sense, so there is an **insert()** method instead. Each insertion will take O(log(n)) time in the size of the set. The last two lines once again use an interator to print out the elements of the set. This time, we have chosen the name of the iterator variable to be j. Its type is now "set< string >::iterator", i.e. an iterator over a set of strings. It is initialized to ss.begin(), which is an iterator referencing the smallest element of the set. The loop runs while j != ss.end(), which is once again an iterator pointing to one after the last element of the set. ++j increments the iterator. *j returns the string being referenced by j. This code will output "`_Delenn_Lennier_Sheridan`".

Here is an easy way to implement tree sort – a sorting algorithm that runs in O(n*log(n)) time. Suppose you have a vector **v** that contains unsorted integers. You can do the following.

Example 9:
```
set< int > s( v.begin(), v.end() );
vector< int > w( s.begin(), s.end() );
```

First, we create a set **s** by passing it a range containing all of the elements of **v**. Then we create another vector, **w**, containing all of the elements of **s**. But, since **s** is a set, the constructor for **w** will read the elements of the set in sorted order. The first line will require O(n*log(n)) time, and the second line is linear time. This is a fairly inefficient way of sorting because it creates 3 instances of each element – one in **v**, one in **s** and one in **w**. That' s 3 times the memory that quicksort would need. We will see a better way to sort vectors soon.

To remove an element from a set, you can use the **erase()** method. It will take an element or an iterator as a parameter – either one will work. To check whether an element is in the set, there is a **count()** method that can be used like this:

Example 10:
```
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 );
}
```

The first loop adds to s all integers between 1900 and 3000, inclusive, that are divisible by 4. The second loop removes from s all integers that are divisible by 100, but are not divisible by 400. s.count (i) is equal to the number of times i appears in the set s, which is ether 0 (if i is not in the set) or 1 (if i is in the set). At the end, s contains all leap years between 1900 and 3000 inclusive. In fact, the second **if** statement is unnecessary because it is always true – s.count(i) always returns 1 in this example.

Maps

```
#include <map>
```

In contrast to all of the previously discussed datastructures that store collections of elements, maps store key-value pairs. The simplest way to visualize a map is to think of a dictionary that for each word (key) gives a description (value). Just as dictionaries are sorted by the words, maps are sorted by the keys. The STL **map** is equivalent to java.util.TreeMap – both are implemented using balanced binary search trees, so insertion, deletion and update operations require **logarithmic** time in the size of the map.

The **map** class has 2 template parameters – key type and value type. To access entries in a map, it is easiest to use the [] (square brackets) operator. Here is how we can assign a number to each day of the week.

Example 11:
```
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;
```

The first line declares a vector of 7 strings. The next 2 lines assign the 7 elements of the vector to be the 7 days of the week. Now for each integer, i, between 0 and 6, i2s[i] is the string representation of the i' ht day of the week. The 4th line declares a map from strings to ints, called s2i. The key type is string, and the value type is int. If you think of the vector as converting from an integer to a string, then the idea is to make the map do the opposite conversion – from a string to the corresponding int. This is done in the last line. For each i, i2s[i] is the name of the day, and we are mapping it to the corresponding integer using the map s2i. As a result, s2i["Sunday"] will be 0, s2["Monday"] will be 1, etc. In C++, the square brackets operator is just like any other function, and it can be redefined or overloaded for different classes. In the **map<string, int>** class, it takes a parameter of type **string** and returns an **int**. In general, it takes a parameter of the same type as the key and returns a reference to the value that is mapped to the given key.

Here is another example of using maps. Suppose that we have a text file and we would like to know the frequency of a certain word – how many times the word appears in the text. We would also like to know how many different words are there in the text. To keep things simple, assume that there is no punctuation and that all words are in lower case letters. Then we can simply use **cin**'s operator >> to read a word into a string.

Example 12a:
```
  string word;
  map< string, int > freq;
  while( cin >> word ) freq[word]++;
```

Once again, we declare a map from strings to ints. We will use it to count how many times a given word appears. In the third line, a few interesting things happen. "while(cin>>word)" will read words into the variable "word", separated by whitespace, until the end of file. For each word read, we will increment freq[word]. But at the beginning freq is empty! freq[word] does not exist yet. It turns out that if you try to access an element of the map that doesn't exist, the map will create one for you. Thus, if the first word we read is "cow", then the statement freq["cow"]++ will first create an entry in freq that maps "cow" to 0 (the default value for an int) and will then increment it to 1. This means that the code above really does work, and at the end of the loop, freq will contain as many different entries as there are different words in the input. Each entry will map a word to a number that is equal to the number of times the word appeared in the input. Now we need a way of getting this information out of the map.

One way to look at a map is as a collection of key-value pairs, and we already know how to extract data from collections – we used iterators. Map iterators are a bit more complicated because they reference key-value pairs instead of just elements. Remember the pair class from the templates example? It had two members, first and second. In this case, first is a key and second is its value. Here is how we can print the frequency table, freq.

Example 12b:
```
  for( map< string, int >::iterator i = freq.begin(); i != freq.end(); ++i )
      cout << (*i).first << ": " << (*i).second << endl;
```

Now we have an iterator over a map from strings to ints (map<string,int>::iterator) called i. It is initialized to freq.begin() and is incremented until it becomes freq.end(). *i is now a key-value pair that has two elements – first (the key) and second (the value). We need to put parentheses around *i because otherwise the dot operator would take priority over the * operator.

Here is an example. Suppose the input was "cat dog cow cow dog pig cow cow". Then code in example 12 would print the following:
```
  cat: 1
  cow: 4
  dog: 2
  pig: 1
```
Note that because the map is sorted by key, the output is sorted lexicographically by the word.

To check whether a given key exists in a map, you can use the **count()** method, just like with a set. You can use **erase(k)** to erase the key-value pair whose key is equal to k. **size()** will return the size of the map. Here is am example of using **count()** and **erase(k)**.

Example 13:
```
map< string, int > freq;
// ... Suppose we initialize freq as in Example 12a
vector< string > commonwords;
commonwords.push_back( "and" );
// ... add more common words to the vector
for ( int i=0; i < commonwords.size(); i++ )
  if ( freq.count( commonwords[i] ) )  // Check if this common word is in the map
    freq.erase( commonwords[i] );      // Erase it so our map don't count it
```

Advanced Features

Sometimes, when using sets and maps, it is necessary to keep the elements sorted by some custom criterion. For example, by default, the < operator on strings performs lexicographic comparison, meaning that a string S is considered less than string T if S would appear before T in a dictionary. But what if we wanted to sort strings by length? In the following example we will do just that – create a set of strings that is sorted first by length and then lexicographically.

Example 14:

```
#include <string>
#include <set>
using namespace std;

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;
}
```

The program will print " cat cow catch copper". The first 3 lines are a standard header. Then we have a declaration of a "struct" called myLessThan. A struct is simply a class whose members are public by default. The struct has one method called "operator()" that takes two parameters – both are references to strings. "const" means that the function is not allowed to modify the strings and the ampersands (&) mean that these are references. s and t are their names. The "const" at the end is saying that the method is not allowed to modify any members of the struct (of which we have none). These syntax have to be there for the custom comparator to work. This method is our custom < operator – it will return true if and only if s is less than t, whatever our definition of "less than" is. First, the method checks if the sizes of the two strings differ; and if they do, it returns true when the size of s is smaller than the size of t. If the sizes are the same, it compares the two strings lexicographically, using the standard < operator.

In the main() function, we create a set of strings and add 4 elements to it. Note that myLessThan is passed as the second template parameter. Finally, we print out the elements of the set in sorted order. Note also that the type of the iterator i is now "set<string,myLessThan>::iterator".

We can use custom comparators in maps as well, by passing the comparator type as the 3rd template parameter, for example "map<string,int,myLessThan> m;".

You can learn a lot more about the STL at http://www.sgi.com/tech/stl.