# Lab 6

*Out: Wednesday 9 March 2005*

## What you'll learn.

Modern C++ comes with a powerful template library, the Standard Template Library, or STL. The STL is based on the independent concepts of algorithms and containers, loosely connected by the concept of iterators. Algorithms access containers only through iterators.

In this lab, you will learn to use many of the STL's algorithms and containers. With such a powerful library finished code is often simpler than the code you would have to write from scratch. Beware though of nasty C++ template errors!

If you missed lecture, please read the lecture notes before attempting this lab. Otherwise, you may find all this talk of templates very confusing.

## How you'll do it.

We provide a set of encrypted text files. Your mission, should you choose to accept it, will be to decrypt all of them automatically.

The files have been encrypted by a random one-to-one mapping of characters to characters. Your job is to retrieve the mapping by computing the frequencies of characters in the text, and comparing them with the frequencies in a "clear" text.

### Problem 6.1 Some STL notes

There are two new links under the links section of the web site. The first one, "Dinkumware C++ Library Reference" points to the online reference of a commercial C++ Library. You need to click on the small paws icon to get to it. The library is compatible with the C++ library that comes with g++ 3.3, the default g++ compiler on our systems.

The second link points to SGI's STL library (which used to be HP's). It is slightly outdated (for example, there are no cool stream iterators described) but you may find it a bit more readable, and it has examples.

We will be using many STL components in this lab. We will go through them one at a time, starting with some functions. All the explanations in this lab assume you have the statement

```
using namespace std;
```

in your program, and that you have included the appropriate headers (which you can find in the documentation). Most of the functions come from the `#include <algorithm>`

header, and most of the containers come from `#include <container_name>` headers (such as `#include <vector>` or `#include <map>`).

# 1 Task 1: Pointers as iterators and `copy`

The major concept of the STL is that of containers accessed by iterators. We'll start slow, with a "container" you know well: an array.

Pointers in arrays can be used as iterators, taking advantage of the built-in *, ++ and = operators. For an array, the address of the array can be used as the begin iterator, and the address of the array plus the size of the array as the end iterator (it points to 'one' past the end of the array).

The first STL function we are going to use is `copy`. To use `copy`, you need to `#include <algorithm>`. `copy` expects three arguments: the begin iterator for the source container, the end iterator for the source container, and the begin iterator for the destination container. The destination container must have enough space to hold all the items to be copied.

**Task:** Create two character arrays of size 27. Put the alphabet in one of them. Use the STL `copy` function to copy the alphabet into the second array. Print both arrays out using `cout`.
Don't get checked off for this one - show it to a TA with the next task.

# 2 Task 2: STL Vectors

STL provides a template class called vector. You create a vector of characters with

```
vector<char> v1;
```

and a vector of Colors with

```
vector<Color> v2;
```

Vectors are initially empty.

The canonical way to add items to them is using the vector method push_back:

```
v1.push\_back(’a’);
```

Item access is like arrays, through a [] operator:

```
v1[0] == ’a’
```

Their begin and end iterators are methods:

```
v1.begin()
v1.end()
```

### Problem 6.1 `back_inserter`

Since vectors are initially empty, we cannot use copy with an empty vector as the destination, since the destination needs to have enough space to hold the items. However, the STL provides a way to create a vector iterator that creates space as needed:

```
back\_inserter(v1)
```

can be used in place of a vector iterator, and will add items to the end of it.

> **Task:** Use copy and back_inserter to copy the alphabet from an array to a vector. It's still not time to get checked off.

### Problem 6.2 `random_shuffle` and `sort`

`random_shuffle` is an STL function that will randomly permute some items in a container. It expects two arguments: the begin and end iterators of the container you want shuffled. [Side note: what happens if you give it a smaller range, like `v.begin() + 1` and `v.end() - 1`?]

`sort` is an STL function that will sort the range you give it (again two iterator arguments). It has a default comparison function, which we will see later how to change.

> **Task:** Shuffle the vector, print it out, sort it, and print it out again.

## 3 Task 3: Reading and writing with iterators

OK, enough containers for a while. Before we proceed any further with them, you should know that iterators are not only good for containers: with operator overloading, you can write iterators that read or write to files!

You can get the equivalent of a begin iterator for an istream by saying

```
istream_iterator<char>(cin)
```

The equivalent of an end iterator is given by

```
istream_iterator<char>()
```

(Note the absence of an argument!) Instead of `<char>` you can create istream_iterators that read in any other type for which `cin >> t` would work.

Output operators are similar. To get the equivalent of a begin iterator, you need to say

```
ostream_iterator<char>(cout, " ")
```

where the " " is used as a separator.

Note that for input iterators we refer to both a begin and an end, while for output iterators (such as the `ostream_iterators` of `back_inserter`) we realy only need a begin, and then we have to make sure that there is enough space for the output.

> **Task:** Use istream iterators on strings, a string vector, and an output iterator to read in `/course/cs034/pub/lab6/hamlet.txt` and output it in a single (very long) line.

# 4 Task 4: Character frequencies

The goal of this lab is to decrypt files that have been encoded with a one to one character substitution, like a = c, c = j, etc. To do this decryption, we operate on the assumption that if we compute the frequency of each character in the encrypted file, and then calculate similar frequencies for a plain text file, we will be able to match up encrypted characters with their plain text partners.

In this task, we are going to use the STL to the max to calculate character frequencies.

### Problem 6.1 read in characters

You will start by building a function that will read all the characters of a file into a string.

```
void read_file(istream& is, string& str);
```

A couple handy hints:

- One way to check if an istream has just failed to read a character (e.g., when it is at the end) is simple something like `if(cin)`. This means that

  ```
  if(cin >> c) {

    foo(c);
  }
  ```

  when c is a character, will attempt to read a character into c, and if it succeeds, foo will be called with that character.

- You can add 'a' to the end of a string str by simply saying str += 'a'.

## Problem 6.2 calculate the frequencies

Next, you need a function that will build a map of frequencies of characters.

```
typedef map<char, int> FreqMap;
void build_freqmap(string& str, FreqMap& f);
```

STL maps work more or less like arrays, except the key (what you index with) can be any type, not just a numeric one. For example, if m is of type `map<int, int>` you can say `m[9] = 0;`, but if m is of type `map<char, string>` you can say `m['t'] = "tee";`.

One note: STL maps cannot distinguish between inserting an item and trying to access an item that is not there. So if m is an empty map, after

```
if (m['t'] < 9)
```

m will have one item, with key 't' and value the default for the value type. You can use m.find('t') to find out if m contained an item with key 't'.

## Problem 6.3 sort the frequencies

Next, you will a function that will copy your map onto a vector. Fortunately, the STL provides such a function,

```
copy(source_begin, source_end, dest_begin),
```

where `source_begin`, `source_end` and `dest_begin` are iterators.

The "elements " of a map are pairs of key,value.

An stl pair is of type `pair<K,V>`. Make your vector of the appropriate template argument. `dest_begin` needs to be adding items to the destination container as it goes along. To do that automatically for vectors, you can use `back_inserter(v)`, where v is a vector.

Once the map is copied to a vector, you need to sort the vector. The relevant stl function is called

```
sort(begin, end).
```

## Problem 6.4 write the characters

Lastly, write a function that will print the characters in order of frequency, by selecting the first item of each pair in the vector.

**Task:** Write the `read_file` and `freq_map` functions as described above; use them together with copy, sort and pair functions to read in a file and print all its characters from most to least frequent. Do this for `/usr/dict/words`, and for `/course/cs034/pub/lab6/hamlet.txt`.

## 5 Task 5: Decryption at last!

With all the work you have done, you are almost ready to decrypt files encrypted with a one-to-one scheme. To do so you need two sorted vectors as you did in the previous task. The first vector sorts the characters by their frequency in the encrypted file, the second vector sorts the characters by their frequency in plain text. The decryption idea is that the most frequent character in the encrypted text represents the most frequent character in the plaintext, etc. If the statistics of the clear text we have correspond those of the encrypted text, the technique should work well.

For your next task, you will compose the two vectors into a single `map<char, char>` that maps encrypted characters into unencrypted characters. You can build this map by walking through the two vectors concurrently, and making a map item for each vector position. The key of the i-th map item is the character in the i-th position of the vector of the encrypted characters, and the value of the i-th map item is the character in the i-th position of the vector of the plain text.

Armed with this map, you can now walk through the encrypted text and decrypt it, character for character. An STL function you might want to use is called `transform`.

## 6 You're Done!