# C and C++

## 8. Standard Template Library (STL)

Stephen Clark

University of Cambridge
(heavily based on last year's notes (Andrew Moore) with thanks to Alastair R. Beresford and Bjarne Stroustrup)

Michaelmas Term 2011

# Standard Template Library - bonus material

**(Templates are examinable; however, STL is NOT examinable for 2011/2012)**

Alexander Stepanov, designer of the Standard Template Library says:

“STL was designed with four fundamental ideas in mind:

- Abstractness
- Efficiency
- Von Neumann computational model
- Value semantics”

It’s an example of generic programming; in other words reusable or “widely adaptable, but still efficient” code

# Advantages of generic programming

- Traditional container libraries place algorithms as member functions of classes
  - Consider, for example, `"test".substring(1,2);` in Java
- So if you have $m$ container types and $n$ algorithms, that's $nm$ pieces of code to write, test and document
- Also, a programmer may have to copy values between container types to execute an algorithm
- The STL does not make algorithms member functions of classes, but uses meta programming to allow programmers to link containers and algorithms in a more flexible way
- This means the library writer only has to produce $n + m$ pieces of code
- The STL, unsurprisingly, uses templates to do this

# Plugging together storage and algorithms

Basic idea:

- define useful data storage components, called containers, to store a set of objects
- define a generic set of access methods, called iterators, to manipulate the values stored in containers of any type
- define a set of algorithms which use containers for storage, but only access data held in them through iterators

The time and space complexity of containers and algorithms is specified in the STL standard

# A simple example

```
#include <iostream>
#include <vector>  //vector<T> template
#include <numeric> //required for accumulate

int  main() {
  int i[] = {1,2,3,4,5};
  std::vector<int> vi(&i[0],&i[5]);

  std::vector<int>::iterator viter;

  for(viter=vi.begin(); viter < vi.end(); ++viter)
    std::cout << *viter << std::endl;

  std::cout << accumulate(vi.begin(),vi.end(),0) << std::endl;
}
```

# Containers

- The STL uses containers to store collections of objects
- Each container allows the programmer to store multiple objects of the same type
- Containers differ in a variety of ways:
  - memory efficiency
  - access time to arbitrary elements
  - arbitrary insertion cost
  - append and prepend cost
  - deletion cost
  - ...

# Containers

- Container examples for storing sequences:
  - `vector<T>`
  - `deque<T>`
  - `list<T>`
- Container examples for storing associations:
  - `set<Key>`
  - `multiset<Key>`
  - `map<Key,T>`
  - `multimap<Key, T>`

# Using containers

```
#include <string>
#include <map>
#include <iostream>

int main() {

  std::map<std::string,std::pair<int,int> > born_award;

  born_award["Perlis"] = std::pair<int,int>(1922,1966);
  born_award["Wilkes"] = std::pair<int,int>(1913,1967);
  born_award["Hamming"] = std::pair<int,int>(1915,1968);
  //Turing Award winners (from Wikipedia)

  std::cout << born_award["Wilkes"].first << std::endl;

  return 0;
}
```

# std::string

- Built-in arrays and the `std::string` hold elements and can be considered as containers in most cases
- You can't call "`.begin()`" on an array however!
- Strings are designed to interact well with C char arrays
- String assignments, like containers, have value semantics:

```
#include <iostream>
#include <string>

int main() {
  char s[] = "A string ";
  std::string str1 = s, str2 = str1;

  str1[0]='a', str2[0]='B';
  std::cout << s << str1 << str2 << std::endl;
  return 0;
}
```

# Iterators

- Containers support iterators, which allow access to values stored in a container
- Iterators have similar semantics to pointers
  - A compiler may represent an iterator as a pointer at run-time
- There are a number of different types of iterator
- Each container supports a subset of possible iterator operations
- Containers have a concept of a `begin`ning and `end`

# Iterator types

| Iterator type | Supported operators |
|---:|---|
| Input | `== != ++ *`(read only) |
| Output | `== != ++ *`(write only) |
| Forward | `== != ++ *` |
| Bidirectional | `== != ++ * --` |
| Random Access | `== != ++ * -- + - += -= < > <= >=` |

- Notice that, with the exception of input and output iterators, the relationship is hierarchical
- Whilst iterators are organised logically in a hierarchy, they do not do so formally through inheritence!
- There are also const iterators which prohibit writing to ref'd objects

# Adaptors

- An adaptor modifies the interface of another component
- For example the `reverse_iterator` modifies the behaviour of an `iterator`

```
#include <vector>
#include <iostream>

int main() {
  int i[] = {1,3,2,2,3,5};
  std::vector<int> v(&i[0],&i[6]);

  for (std::vector<int>::reverse_iterator i = v.rbegin();
       i != v.rend(); ++i)
    std::cout << *i << std::endl;

  return 0;
}
```

# Generic algorithms

- Generic algorithms make use of iterators to access data in a container
- This means an algorithm need only be written once, yet it can function on containers of many different types
- When implementing an algorithm, the library writer tries to use the most restrictive form of iterator, where practical
- Some algorithms (e.g. `sort`) cannot be written efficiently using anything other than random access iterators
- Other algorithms (e.g. `find`) can be written efficiently using only input iterators
- Lesson: use common sense when deciding what types of iterator to support
- Lesson: if a container type doesn't support the algorithm you want, you are probably using the wrong container type!

# Algorithm example

- Algorithms usually take a `start` and `finish` iterator and assume the valid range is `start` to `finish-1`; if this isn't true the result is undefined

Here is an example routine `search` to find the first element of a storage container which contains the value `element`:

```
//search: similar to std::find
template<class I,class T> I search(I start, I finish, T element)
  while (*start != element && start != finish)
    ++start;
  return start;
}
```

# Algorithm example

```
#include "example23.hh"

#include "example23a.cc"

int main() {
  char s[] = "The quick brown fox jumps over the lazy dog";
  std::cout << search(&s[0],&s[strlen(s)],'d') << std::endl;

  int i[] = {1,2,3,4,5};
  std::vector<int> v(&i[0],&i[5]);
  std::cout << search(v.begin(),v.end(),3)-v.begin()
            << std::endl;

  std::list<int> l(&i[0],&i[5]);
  std::cout << (search(l.begin(),l.end(),4)!=l.end())
            << std::endl;

  return 0;
}
```

# Heterogeneity of iterators

```
#include "example24.hh"

int main() {
  char one[] = {1,2,3,4,5};
  int two[] = {0,2,4,6,8};
  std::list<int> l (&two[0],&two[5]);
  std::deque<long> d(10);

  std::merge(&one[0],&one[5],l.begin(),l.end(),d.begin());

  for(std::deque<long>::iterator i=d.begin(); i!=d.end(); ++i)
    std::cout << *i << " ";
  std::cout << std::endl;

  return 0;
}
```

# Function objects

- C++ allows the function call "`()`" to be overloaded
- This is useful if we want to pass functions as parameters in the STL
- More flexible than function pointers, since we can store per-instance object state inside the function
- Example:

```
struct binaccum {
  int operator()(int x, int y) const {return 2*x + y;}
};
```

# Higher-order functions in C++

- In ML we can write: `foldl (fn (y,x) => 2*x+y) 0 [1,1,0];`
- Or in Python: `reduce(lambda x,y: 2*x+y, [1,1,0])`
- Or in C++:

```cpp
#include<iostream>
#include<numeric>
#include<vector>

#include "example27a.cc"

int main() { //equivalent to foldl

  bool binary[] = {true,true,false};
  std::cout<< std::accumulate(&binary[0],&binary[3],0,binaccum())
            << std::endl; //output: 6

  return 0;
}
```

# Higher-order functions in C++

- By using reverse iterators, we can also get foldr:

```
#include<iostream>
#include<numeric>
#include<vector>

#include "example27a.cc"

int main() { //equivalent to foldr

  bool binary[] = {true,true,false};
  std::vector<bool> v(&binary[0],&binary[3]);

  std::cout << std::accumulate(v.rbegin(),v.rend(),0,binaccum());
  std::cout << std::endl; //output: 3

  return 0;
}
```