Standard Template Library STL

- What are templates and STL and how to use them?
- Some common data-structures
- Comparator functions
- Some more datastructures
- Iterators
- Algorithms (sort, find, reverse, ...)
- Other templated types (pair, complex, string and rope)
- Efficiencies of the common Data-Structures
- How to use the STL docs

In the beginning..

- Say I want to create a queue of integers
  - Good fine, after a bit of work this can be done
- Now I want a queue of strings...
  - ok... remake the queue but with strings this time
- Wait? Haven't I just done 2 times the work?
  - Yes... yes I have
  - Wouldn't it be nice if you could just do it once...
    - Yes... yes it would :)

Introducing Templates

- Templates are a way of creating a datastructure once so that it can assigned any arbitrary datatype after
- Similar to generic types in Java (but NOT the same)
- E.g. We create a template queue once
    - Now we can easily use a string version and an integer version without having to recode it.

Introducing The STL

- Turns out you won't actually have to code any template classes yourself anyway
  - It's all been done for you
- The Standard Template Library:
  - A library of standard templates
    - vectors, queues, priority_queues, sets, maps ... etc etc etc
  - Very powerful
  - Very fast
  - Very flexible

Templates in C++: Syntax

- STL has the vector<T> templated class
- If we want a vector of ints we simply use:
  - vector<int> my_integers;
- Or for doubles
  - vector<double> my_doubles;
- Or for any class you want
  - class my_node_class {int n; double weight; ...};
  - vector<my_node_class> my_nodes;

Why STL is fantastic!

- Fast – optimised to hell and back!
  - All templates are made at compile time
    - It's as if someone quickly codes up the specific data-structure for you just before compiling
    - Unlike Java (which does it at run-time ... very very slow)
- Powerful and Vast
  - Easy to make any datastructure of any type
  - Can also make arrays of templated types
  - vector<int> [N]; (you can't do this in Java!)
  - There are more then enough datastructures to suit any situation
- And it's all done for you!!!!

Common Data-Structures

- Vector
- List
- Queue
- Stack
- Map
- Priority Queue
- Set
- Hashes
- ...

Sequences

- List - #include <list>
  - Your standard issue linked list (doubly linked)
    - list<double> my_list;
    - my_list.push_back(42); // adds to back of array
    - my_list.push_front(21); // adds to front of array
    - double d = my_list.back(); // gets item at back of list
    - double d2 = my_list.front(); // gets item at front of list
    - my_list.pop_back(); // removes back item from list
    - my_list.pop_front(); // removes front item from list
  - Can also insert in the middle (explained a bit later)

Sequences

- Vector - #include <vector>
  - Resizeable array
    - vector<int> my_vec; // array size 0
    - vector<int> my_vec(100); // array size 100
  - Has same operations as list
    - push_back(), push_front(), front(), back() ...
  - Can use standard [] notation (operator overloading!)
    - my_vec[3] = 11;
    - int my_int = my_vec[9];

Queues and Stacks

- Queue - #include <queue>
  - queue<double> Q; // empty queue
  - Q.push_back(3.14);
  - Q.push_back(2.7)
  - double d = Q.top(); // will be 3.14
  - Q.pop(); // removes the top element from the queue
- Stack - #include <stack>
  - Works in the same way as queue, except FIFO

Sorted data-structures

- These datastructures require some form or order
  - Either have to give it a less-then function or define the less-then operator (<)
  - operator< already defined for int, double, float etc
  - Can define it for whatever class you want

```
class my_class {
    int a, b; double c;
    bool operator<(const my_class& m) const {
        return c < m.c;}   };
```

  - Can also define the == operator similarly

Maps

- Map - #include <map>
  - Maps one type to another - map<key, data>
    - map<int, string> my_map;
    - my_map[1] = ”One”;
    - String s = my_map[1]; // will be ”One”
    - String s2 = my_map[3]; // will be default value of ””
  - Can map anything to anything
    - map<string, int> string_map;
    - string_map[”January”] = 31;

Priority Queue

- Priority Queue - #include <queue>
  - Must have operator< defined
  - priority_queue<int> P;
  - Same commands as a normal queue
    - P.push(), P.top(), P.pop()
  - Except top will return the 'largest' value
    - Depending on how you define large
  - If you want the smallest value
    - Define large to be small ;)
      - return return c > m.c;

General functions

- By now you would have seen that some functions are common to nearly all structures
  - .size() returns the number of elements
  - .empty() returns whether there are elements at all
    - Rather use .empty() instead of .size() == 0
    - Since .size() might not be O(1) - can anyone say list?
  - You've already seen front(), back() push_back() etc...
    - These are common to most structures (not all)
    - Check the docs if you are unsure

Iterators

- Having a structure is great
- But what if you want to go through all the elements of a structure?
- Use iterators!
- Almost all STL data-structures have iterators
  - Like priority_queues don't have iterators

Iterators: Example

```
vector<my_class> my_vec;
... // adding stuff to my_vec
for (vector<my_class>::iterator i = my_vec.begin() ; i != my_vec.end() ; i++)
{
    // note! It is *i, not i (the astrik dereferences the iterator)
    cout << *i << endl;
    cout << (*i).a << endl;
    cout << i->a << endl; // -> just a shorthand way of writing (*i).
}
```

- Can do this with list, set, queue, stack...
  - Check documentation for more info

Whats going on here!?

- vector<my_class>::iterator i = my_vec.begin()
  - Like int i = 0;
- i++
  - This is like i.iterate() or i.next(). Just move on to the next element
- i != my_vec.end()
  - my_vec.end() points to the position just after the last element in the datastructure

Iterators: my_vec.end();

- Why do we say != instead of < ??
  - There is no sense of less then in an iterator.
- Say we are at the last element of the list:
  - i++ will then make i point to the position just after the list
  - the position just after the list == my_vec.end()
  - Also useful as a 'NULL' value (c.f. algorithms...)

Other Iterator stuff

- Some iterators are bidirectional (i.e. can use i--)
- Reverse iterators
  - Backward traversal of a list
  - For(list<int>::reverse_iterator i = my_list.r_begin() ; i != my_list.r_end(); i–)
- For vectors:
  - [] operator is slightly slower then useing iterators

Now that you know iterators...

- list<int>::iterator i; // and i is in the middle of the list

insert(iterator pos, const T& x)

- my_list.insert(i, 45); // inserts 45 just before i
  - Same for vectors
- my_list.erase(i); // erases element at i
  - But what if you have this

```
for (list<int>::iterator i = my_list.begin(); i !=
  my_list.end() ; i++) {
    if (*i == 45)
        my_list.erase(i);
}
```

Erasing elements

```
for (list<int>::iterator i = my_list.begin(); i !=
  my_list.end() ; i++) {
    if (*i == 45)
        my_list.erase(i);
}
```

- The item at i will be erased
- When the next loop comes around, i++ will be called
- But we just deleted i !

```
for (list<int>::iterator i = my_list.begin(); i !=
  my_list.end() ; i++) {
    if (*i == 45)
        my_list.erase(i--); // problem solved
}
```

Sets

- Set - #include<set>
  - Unique Sorted set of elements
    - So no two elements will be the same
  - Must have operator< defined
    - Since iterator will run through them in order
  - set<double> my_set;
  - my_set.insert(3.1459);
  - my_set.remove(11.5);

Set iterators

- upper and lower bounds of a set
  - lower_bound
  - set<point>:iterator = my_set.lower_bound(10);
    - Returns the first element that is >= 10
  - set<point>:iterator = my_set.upper_bound(90);
    - Returns the first element that is <= 90
  - So a set {1, 4, 15, 39, 89, 90, 102, 148}
    - my_set.lower_bound(10); //will point to 4
    - my_set.upper_bound(90); //will point to 90

Hash Sets

- Hash Set - #include <ext/hash_set>
- using namespace __gnu_cxx;
- hash_set<const char *> my_hash_set;
- my_hash_set.insert(”a string”);
- my_hash_set.insert(”another string”);
- my_hash_set.find(”a string”); // returns an iterator
  - Returns my_hash_set.end() if not found

Hash Map

- Hash Map - #include <ext/hash_map>
- using namespace __gnu_cxx;
- Like a map
  - hash_map<int, const char *> my_hash_map;
  - my_hash_map[3] = ”a string”;

The Hashing Function

- As you know the hash set and hash map need a hashing function
- This is already defined for int, double, float, char byte, short, long and const char *
- If you use your own class then you have to provide your own hashing function
  - Use function objects (explained later)
  - hash_set<my_class, my_hash_func> my_hash_set;

Algorithms

- We have this lovely general way of using data-structures:
- Why don't we use them to write general algorithms?
  - We do! (by ”we” I mean the people who wrote STL)
- sort(), find(), unique(), count(), reverse() are all general algorithms at your disposal
  - There are others...
- #include <algorithm>

Algorithms: Types

- Algorithms can loosely be group into 2 categories
  - Data Transformation: These algorithms transform your data by applying operations on them. Can overwrite the original or copy to a new container. eg: reversing, sorting, etc
  - Data Information: These algorithms retrieve information about your data. eg: minimum element, searching, etc

Algorithms: Before we begin

- A lot of algorithms use function objects.
- Function objects are just objects of classes that have the ( ) operator overloaded.
- Function objects must have the correct parameters for your program to compile.
- Can often be interchangable with functions themselves.

Algorithms: Before we begin

- This is legal
  - vector<double> my_vec;
  - sort(my_vec.begin(), my_vec.end());
- And so is this
  - double my_arr[N];
  - sort(my_arr, my_arr+N);

Algorithms: Transformations

- copy(myArr, myArr+N, myVec.begin());
- copy_n(myArr, N, myVec.begin());
- copy_backward(myArr, myArr+N, myVec.end());
  - Copies data from one place in memory to another.
  - Can specify iterators for the range to copy or specify a iterator to the beginning of a range.
  - Usually copies from start to end, but can do the other way.

Algorithms: Transformations

- swap(a, b);
  - Swaps two values.
- iter_swap(myArr+3, myArr+4);
  - Swaps two values of iterators.
- swap_ranges(myArr+1, myArr+N/2, myArr+1+N/2);
  - Swaps two ranges specified by the beginning and end of the first range and the beginning of the second.

Algorithms: Transformations

- transform(myArr, myArr+N, myVec.begin(), fabs)
  - Transforms all the elements in the range specified by the first two iterators and stores the result in the third iterator. The last parameter is a unary function object or function giving the result of the transformation.
- transform(myArr, myArr+N, myVec.begin(), myVec.begin(), pow)
  - Same as above, except with a binary function. Need to specify an extra iterator to the beginning of a second range.

Algorithms: Transformations

- fill(myArr, myArr+N. setValue);
  - Sets all values in the range of the first two iterators to the set value.
- fill_n(myArr, N, setValue);
  - Same as above, but can specify exactly how many elements to fill.
- generate(myArr, myArr+N, functionObject);
- generate_n(myArr, N, functionObject);
  - Same as the above, but can specify a function object that takes no arguments to get a value to fill each element.

Algorithms: Transformations

- unique(myArr, myArr+N);
  - Removes consecutive duplicate items specified by the range.
- unique(myArr, myArr+N, binaryPredicate);
  - Removes consecutive duplicate items specified by the range, and using the binary predicate to test for equality.
  - Does NOT remove all duplicates in a range, however, if the range is sorted, all duplicates in that range will be removed.
  - Also copy versions.

Algorithms: Transformations

- reverse(myArr, myArr+N);
  - Reverses the range specified by the iterator.
  - Also a copy version to store the reversed range in a new container.

Algorithms: Transformations

- sort(myArr, myArr+N);
  - Sorts the range specified.
  - Uses the < operator to compare elements.
  - Guaranteed O(Nlog(N)). Uses a introsort.
- stable_sort(myArr, myArr+N);
  - Same as above, but is stable.
- Separate sort functions for linked lists.

Algorithms: Transformations

- A few others functions for transforming data.
  - Statistical functions for finding random samples and shuffling the data.
  - Mathematical functions for finding unions, intersections, etc of sets.
  - Functions for finding permutations of your set.
  - Functions for find the n-th 'smallest' element in your set.

Algorithms: Information

- find(myArr, myArr+N, findValue);
  - Performs a linear search on the range. Returns the first iterator such that the value at that iterator is equal to findValue.
- find_if(myArr, myArr+N, predicate);
  - Same as above, but instead of testing for equality with a specific element, it tests for truth of a predicate.
- Also find_first_of which searches for the first of a list of values in the range.

Algorithms: Information

- lower_bound(myArr, myArr+N, findValue);
  - Performs a binary search to return an iterator to the first appearance of findValue in the range.
- upper_bound(myArr, myArr+N, findValue);
  - Same as above, but returns an iterator to 'one past' the last element equal to findValue.
- equal_range(myArr, myArr+N, findValue);
  - Same as above, but returns a pair of iterators representing the range on which all values equal findValue.

Algorithms: Information

- binary_search(myArr, myArr+N, findValue)
  - Returns true if the findValue is in the range specified by the iterators and false otherwise.
- All four of the binary search functions can also take comparators.
- Reminder: Comparators are binary predicates, ie: function objects which take two objects and return a boolean value.

Algorithms: Information

- Several other functions that can be used to get information.
  - Mathematical functions that allow you to calculate the minimum and maximum of sets, sum of elements, etc.

Other templated types

- pair<T, Y>
- basically two objects lumped together
  - e.g. pair<int, double> could represent an index and an associated weight
  - can have a pair<double, pair<int,int> >
    - represents a weight and two nodes (perhaps...)
  - pair<double, pair<int,int>>; WRONG!!!!
    - c++ gets confused with the >> operator (just use a space)
  - Comparisons compare first, then second.

Accessing pairs

- to access elements in the pair:
  - pair <string, int> my_pair;
    - my_pair.first = ”a string”;
    - my_pair.second = 5;
    - my_pair = make_pair(”another string”, 42);
- Can have arrays of pairs
  - pair <int, int> edges [N];
    - edges[5].first = 64;

Complex numbers

- complex - #include<complex>
  - Can be treated like a pair of numbers (x,y),
    - but with certain mathematical functions that are quite useful
  - complex<double> coord; // vector
    - Typically complex<T> can be treated as a handy built-in 2D vector class.
  - A = a + bi, conj(A) = a – bi
  - real(A) = a, imag(A) = b
  - conj(A)xB = **A**.**B** + (**A**x**B**)i

String and Rope

- STL provides two data structures for character strings.
  - string
    - Your normal familiar string.
    - Provides functions like substring, length, etc.
    - Provides functions for getting the underlying string data.
  - rope
    - Not your normal familiar string.
    - Better than strings in certain circumstances, however more complicated and unnecessary. Different semantics to string.

Efficiencies

- unsorted array

Insert at front	O(N)
Insert in middle	O(N)
Insert at end	O(1)
Remove at front	O(1)
Remove in middle	O(N)
Remove at end	O(1)
Find element	O(N)
Find minimum	O(N)
Goto N'th item	O(1)

Efficiencies

- sorted array

Insert at front	O(N)
Insert in middle	O(N)
Insert at end	O(N)
Remove at front	O(1)
Remove in middle	O(N)
Remove at end	O(1)
Find element	O(log(N))
Find minimum	O(1)
Goto N'th item	O(1)

Efficiencies

- list

Insert at front	O(1)
Insert in middle	O(1)
Insert at end	O(1)
Remove at front	O(1)
Remove in middle	O(1)
Remove at end	O(1)
Find element	O(N)
Find minimum	O(N)
Goto N'th item	O(N)

Efficiencies

- vector

Insert at front	O(N)
Insert in middle	O(N)
Insert at end	O(1)
Remove at front	O(1)
Remove in middle	O(N)
Remove at end	O(1)
Find element	O(N)
Find minimum	O(N)
Goto N'th item	O(1)

Efficiencies

- queue

Insert	O(1)
Remove	O(1)

Efficiencies

- stack

Insert	O(1)
Remove	O(1)

Efficiencies

- priority_queue

Insert	O(1)
Remove	O(log(N))
Find minimum	O(1)

Efficiencies

- set

Insert	O(log(N))
Remove	O(log(N))
Find element	O(log(N))
Find minimum	O(1)

Efficiencies

- map

Insert	O(log(N))
Remove	O(log(N))
Find element	O(log(N))

Efficiencies

- hash_set

Insert	O(1)
Remove	O(1)
Find element	O(1)
Find minimum	O(N)

Efficiencies

- hash_map

Insert	O(1)
Remove	O(1)
Find element	O(1)

How to use the STL docs

- The STL documentation is all encompassing
  - will tell you everything you need to know
- but!
  - Horrible to read
- So we're going to show you how...
    - goto stl_docs;
      - ...