EECS 211
STL Summary

Home Lectures
Class Info Newsgroup
Links Assignmen

This is a brief summary of the containers in the C++ Standard Library (once informally known as the Standard Template Library or STL). It deliberately sacrifices completeness for simplicity. Information is also available on the iterators and algorithms.

In my opinion, the best overall reference to the Standard Library is Josuttis' The C++ Standard Library.

The containers described below are:

- vectors
- lists
- deques
- stacks
- queues
- priority queues
- sets and multisets
- maps and multimaps
- pairs

For each operation, there is a guaranteed upper-bound on the complexity. See Section 19.2.1 of Deitel for a brief description of "Big Oh" notation.

Vector

Header

```
#include <vector>
```

Constructors

`vector<T> v;`	Make an empty vector.	O(1)
`vector<T> v(n);`	Make a vector with N elements.	O(n)
`vector<T> v(n, value);`	Make a vector with N elements, initialized to value.	O(n)
`vector<T> v(begin, end);`	Make a vector and copy the elements from `begin` to `end`.	O(n)

Accessors

`v[i]`	Return (or set) the I'th element.	O(1)
`v.at(i)`	Return (or set) the I'th element, with bounds checking.	O(1)

`v.size()`	Return current number of elements.	O(1)
`v.empty()`	Return true if vector is empty.	O(1)
`v.begin()`	Return random access iterator to start.	O(1)
`v.end()`	Return random access iterator to end.	O(1)
`v.front()`	Return the first element.	O(1)
`v.back()`	Return the last element.	O(1)
`v.capacity()`	Return maximum number of elements.	O(1)

Modifiers

`v.push_back(value)`	Add value to end.	O(1) (amortized)
`v.insert(iterator, value)`	Insert value at the position indexed by iterator.	O(n)
`v.pop_back()`	Remove value from end.	O(1)
`v.erase(iterator)`	Erase value indexed by iterator.	O(n)
`v.erase(begin, end)`	Erase the elements from `begin` to `end`.	O(n)

Deque

Header

```
#include <deque>
```

Constructors

`deque<T> d;`	Make an empty deque.	O(1)
`deque<T> d(n);`	Make a deque with N elements.	O(n)
`deque<T> d(n, value);`	Make a deque with N elements, initialized to value.	O(n)
`deque<T> d(begin, end);`	Make a deque and copy the values from `begin` to `end`.	O(n)

Accessors

`d[i]`	Return (or set) the I'th element.	O(1)
`d.at(i)`	Return (or set) the I'th element, with bounds checking.	O(1)
`d.size()`	Return current number of elements.	O(1)
`d.empty()`	Return true if deque is empty.	O(1)
`d.begin()`	Return random access iterator to start.	O(1)
`d.end()`	Return random access iterator to end.	O(1)
`d.front()`	Return the first element.	O(1)
`d.back()`	Return the last element.	O(1)

Modifiers

`d.push_front(value)`	Add value to front.	O(1) (amortized)
`d.push_back(value)`	Add value to end.	O(1) (amortized)
`d.insert(iterator, value)`	Insert value at the position indexed by iterator.	O(n)
`d.pop_front()`	Remove value from front.	O(1)
`d.pop_back()`	Remove value from end.	O(1)
`d.erase(iterator)`	Erase value indexed by iterator.	O(n)
`d.erase(begin, end)`	Erase the elements from `begin` to `end`.	O(n)

List

Header

```
#include <list>
```

Constructors

`list<T> l;`	Make an empty list.	O(1)
`list<T> l(begin, end);`	Make a list and copy the values from `begin` to `end`.	O(n)

Accessors

`l.size()`	Return current number of elements.	O(1)
`l.empty()`	Return true if list is empty.	O(1)
`l.begin()`	Return bidirectional iterator to start.	O(1)
`l.end()`	Return bidirectional iterator to end.	O(1)
`l.front()`	Return the first element.	O(1)
`l.back()`	Return the last element.	O(1)

Modifiers

`l.push_front(value)`	Add value to front.	O(1)
`l.push_back(value)`	Add value to end.	O(1)
`l.insert(iterator, value)`	Insert value after position indexed by iterator.	O(1)
`l.pop_front()`	Remove value from front.	O(1)
`l.pop_back()`	Remove value from end.	O(1)
`l.erase(iterator)`	Erase value indexed by iterator.	O(1)
`l.erase(begin, end)`	Erase the elements from `begin` to `end`.	O(1)
`l.remove(value)`	Remove all occurrences of value.	O(n)
`l.remove_if(test)`	Remove all element that satisfy test.	O(n)
`l.reverse()`	Reverse the list.	O(n)

`l.sort()`	Sort the list.	O(n log n)
`l.sort(comparison)`	Sort with comparison function.	O(n logn)
`l.merge(l2)`	Merge sorted lists.	O(n)

Stack

In the C++ STL, a stack is a *container adaptor*. That means there is no primitive stack data structure. Instead, you create a stack from another container, like a list, and the stack's basic operations will be implemented using the underlying container's operations.

Header

```
#include <stack>
```

Constructors

`stack<T> s;`	Make an empty stack using a deque.	O(1)
`stack<T, container<T> > s;`	Make an empty stack using the given container.	O(1)

Accessors

`s.top()`	Return the top element.	O(1)
`s.size()`	Return current number of elements.	O(1)
`s.empty()`	Return true if stack is empty.	O(1)

Modifiers

`s.push(value)`	Push value on top.	Same as `push_back()` for underlying container.
`s.pop()`	Pop value from top.	O(1)

Queue

In the C++ STL, a queue is a *container adaptor*. That means there is no primitive queue data structure. Instead, you create a queue from another container, like a list, and the queue's basic operations will be implemented using the underlying container's operations.

Don't confuse a `queue` with a `deque` or a `priority_queue`.

Header

```
#include <queue>
```

Constructors

`queue<T> q;`	Make an queue stack using a deque.	O(1)
`queue<T, container<T> > q;`	Make an empty queue using the given container.	O(1)

Accessors

`q.front()`	Return the front element.	O(1)
`q.back()`	Return the rear element.	O(1)
`q.size()`	Return current number of elements.	O(1)
`q.empty()`	Return true if queue is empty.	O(1)

Modifiers

`q.push(value)`	Add value to end.	Same for `push_back()` for underlying container.
`q.pop()`	Remove value from front.	O(1)

Priority Queue

In the C++ STL, a priority queue is a *container adaptor*. That means there is no primitive priorty queue data structure. Instead, you create a priority queue from another container, like a deque, and the priority queue's basic operations will be implemented using the underlying container's operations.

Priority queues are neither first-in-first-out nor last-in-first-out. You push objects onto the priority queue. The top element is always the "biggest" of the elements currently in the priority queue. Biggest is determined by the comparison predicate you give the priority queue constructor.

> If that predicate is a "less than" type predicate, then biggest means largest.
>
> If it is a "greater than" type predicate, then biggest means smallest.

Header

`#include <queue>` -- not a typo!

Constructors

`priority_queue<T, container<T>, comparison<T> > q;`	Make an empty priority queue using the given container to hold values, and comparison to compare values. container defaults to `vector<T>` and comparison defaults to `less<T>`.	O(1)

Accessors

`q.top()`	Return the "biggest" element.	O(1)
`q.size()`	Return current number of elements.	O(1)
`q.empty()`	Return true if priority queue is empty.	O(1)

Modifiers

`q.push(value)`	Add value to priority queue.	O(log n)
`q.pop()`	Remove biggest value.	O(log n)

Set and Multiset

Sets store objects and automatically keep them sorted and quick to find. In a `set`, there is only one copy of each object. If you try to add another equal object, nothing happens. `multiset`s are declared and used the same as `set`s but allow duplicate elements.

Sets are implemented with balanced binary search trees, typically red-black trees. Thus, they provide logarithmic storage and retrieval times. Because they use search trees, sets need a comparison predicate to sort the keys. `operator<()` will be used by default if none is specified a construction time.

> *In a set, one object is considered equal to another if it is neither less than nor greater than the other object. `operator==()` is not used.*

Header

```
#include <set>
```

Constructors

`set< type, compare > s;`	Make an empty set. `compare` should be a binary predicate for ordering the set. It's optional and will default to a function that uses `operator<`.	O(1)
`set< type, compare > s(begin, end);`	Make a set and copy the values from `begin` to `end`.	O(n log n)

Accessors

`s.find(key)`	Return an iterator pointing to an occurrence of `key` in `s`, or `s.end()` if `key` is not in `s`.	O(log n)
`s.lower_bound(key)`	Return an iterator pointing to the first occurrence of `key` in `s`, or `s.end()` if key is not in `s`.	O(log n)
`s.upper_bound(key)`	Return an iterator pointing to the first occurrence of an item greater than `key` in `s`, or `s.end()` if no such item is found.	O(log n)
`s.equal_range(key)`	Returns a pair of `lower_bound(key)` and `upper_bound(key)`.	O(log n)

`s.count(key)`	Returns the number of items equal to `key` in `s`.	O(log n)
`s.size()`	Return current number of elements.	O(1)
`s.empty()`	Return true if set is empty.	O(1)
`s.begin()`	Return an iterator pointing to the first element.	O(1)
`s.end()`	Return an iterator pointing one past the last element.	O(1)

Modifiers

`s.insert(iterator, key)`	Inserts `key` into `s`. `iterator` is taken as a "hint" but `key` will go in the correct position no matter what. Returns an iterator pointing to where `key` went.	O(log n)
`s.insert(key)`	Inserts `key` into `s` and returns a pair `p` where `p.first` is an iterator pointing to where `key` was stored, and `p.second` is true if `key` was actually inserted, i.e., was not already in the set.	O(log n)

Map and Multimap

Maps can be thought of as generalized vectors. They allow `map[key] = value` for any kind of `key`, not just integers. Maps are often called associative tables in other languages, and are incredibly useful. They're even useful when the keys are integers, if you have very sparse arrays, i.e., arrays where almost all elements are one value, usually 0.

Maps are implemented as sets of pairs of keys and values. The pairs are sorted based on the keys. Thus, they provide logarithmic storage and retrieval times, but require a comparison predicate for the keys. `operator<()` will be used by default if none is specified a construction time.

Map types are a bit complicated because of the pairs, so it's best to use **typedef** to create more readable type names, like this:

```
typedef map<string, double> ValueMap;
typedef ValueMap::value_type VMPair;
typedef ValueMap::iterator VMIterator;
```

Definitions like the above will make `find()` and `insert()` a lot simpler:

```
ValueMap vm;
vm[ "abc" ] = 2.0;
vm[ "def" ] = 3.2;
vm.insert( VMPair( "ghi", 6.7 ) );

VMIterator iter = vm.find( "def" );
if ( iter != vm.end() ) {
  cout << "Value of " << iter->first " << " is " << iter->second << endl;
}
```

You can just use `map[key]` to get the value directly without an iterator.

> *Warning: `map[key]` creates a dummy entry for `key` if one wasn't in the map*

before. Sometimes, that's just what you want. When it isn't, use `find()`.

`multimap`s are like `map` except that they allow duplicate keys. `map[key]` is not defined for multimaps. Instead you must use `insert()` to add entry pairs, and `find()`, or `lower_bound()` and `upper_bound()`, or `equal_range()` to retrieve entry pairs.

Header

```
#include <map>
```

Constructors

`map< key_type, value_type, key_compare > m;`	Make an empty map. `key_compare` should be a binary predicate for ordering the keys. It's optional and will default to a function that uses `operator<`.	O(1)
`map< key_type, value_type, key_compare > m(begin, end);`	Make a map and copy the values from `begin` to `end`.	O(n log n)

Accessors

`m[key]`	Return the value stored for `key`. This adds a default value if `key` not in map.	O(log n)
`m.find(key)`	Return an iterator pointing to a key-value pair, or `m.end()` if key is not in map.	O(log n)
`m.lower_bound(key)`	Return an iterator pointing to the first pair containing key, or `m.end()` if key is not in map.	O(log n)
`m.upper_bound(key)`	Return an iterator pointing one past the last pair containing key, or `m.end()` if key is not in map.	O(log n)
`m.equal_range(key)`	Return a `pair` containing the lower and upper bounds for key. This may be more efficient than calling those functions separately.	O(log n)
`m.size()`	Return current number of elements.	O(1)
`m.empty()`	Return true if map is empty.	O(1)
`m.begin()`	Return an iterator pointing to the first pair.	O(1)
`m.end()`	Return an iterator pointing one past the last pair.	O(1)

Modifiers

`m[key] = value`	Store `value` under `key` in map.	O(log n)
`m.insert(pair)`	Inserts the <key, value> pair into the map. Equivalent to the above operation.	O(log n)

Pair

A pair is a bit like a Lisp CONS cell. It holds just two values. They can be different types. For simplicity, pairs are simple generic `struct`'s with two public data members: `first` and `second` and a simple constructor that takes the two values to store.

Header

```
#include <utility>
```

Constructors

`pair< first_type, second_type > p(` *`first, second`* `);`	Makes a pair. Both values must be given.	O(1)
`pair< first_type, second_type > p(` *`pair`* `);`	Makes a pair from another pair.	O(1)

Accessors

`p.first`	Returns the first value in the pair.	O(1)
`p.second`	Returns the second value in the pair.	O(1)

Modifiers

There are no modifiers.

Comments? Send mail to c-riesbeck@northwestern.edu.