Data Structures in the STL

The reason you are using C++ over C and Pascal

Keegan Carruthers-Smith

1st Training Camp 2011
South African Computer Olympiad
Department of Computer Science
University of Cape Town

29 April 2011

Before we begin

- All of the following are in the std namespace, so you need to put "using namespace std;" somewhere or refer to things like "std::vector"
- There is more info available to you during the contest in the STL documentation. Go to `http://olympiad.cs.uct.ac.za/docs/` there should be a link somewhere. A nice list is under the "Table of Contents" section for STL.

Vector

```
#include <vector>
vector<int> vi; // empty vector
vector<float> vf(10); // 10 floats all 0
vector<char> vc(30, 'a'); // 30 chars all 'a'

vi.push_back(3); // Add 3 to the end of the vector
vi[0]; // Returns 3

// vi.size() == 0. Everything except vi[0] is 0
vi.resize(100);

// Make values from index 10 to 19 random values
generate(vi.begin() + 10, vi.begin() + 20, rand);

sort(vi.begin(), vi.end()); // sort the vector
vi.pop_back(); // Removes last element. Does not
    return it.
```

Vector
Niceties

- $O(1)$ random access
- $O(1)$ insertion and removal at the back
- Template specialization for `vector<bool>`. Packs bits into chars efficiently

Vector
Pitfalls

- $O(n)$ insertion and removal anywhere but the back.
- `[]` operator still fails silently if index is out of bounds. If you want `ArrayIndexOutOfBoundsException` like in Java, you can use `vector.at(index)`
- The difference between `vector.reserve(size)` and `vector.resize(size)` is subtle. `reserve` only allocates space, but does not initialize the elements. Useful when you know there are going to be N elements, and then you can just use `push_back`
- Be careful when using iterators and mutating the vector.

Deque
Vector in disguise

Exactly the same as a vector except for:

- $O(1)$ insertion and removal from the front of the container.
- No `reserve` or `resize`.

Deque

Slidings windows made easy

```
#include <deque>
deque< pair<int, int> > w;

// Inserting x at time t
while (!w.empty() && x <= w.back().first)
  w.pop_back();
w.push_back(make_pair(x, t));

// Removing elements that are not in the window
while (!w.empty() && T <= w.front().second)
  w.pop_front();

// Smallest element in window
w.front().first;
```

A Detour through iterators

```
vector<int> v(10);
generate(v.begin(), v.end(), 10);
vector<int>::iterator it;
for (it = v.begin(); it != v.end(); ++it) {
  cout << *it << endl;
}
```

- Iterator is an abstraction of a pointer.
- Replace `vector<int>` with nearly any container described in these slides and the code will still work.
- `it != v.end()` not `it < v.end()`
- `*it` is the value that `it` points to
- `++it` is faster than `it++`

List

- `#include <list>`
- Doubly Linked List
- $O(1)$ insertion, deletion, access once you have an iterator.
- $O(n)$ time to get the iterator.
- Supports $O(1)$ increment and decrement of the iterator. If you just need increment `#include <slist>` for a Singly Linked List.

List

Why use this instead of a Vector

Some algorithms for free that are fast

```
// Create list<int> a = [1, 2, 3, 3, 2, 4, 4]
// Create list<int> b = [2, 4, 4, 6, 5, 6, 7]
// Let list<int>::iterator it point to 5 in b

b.remove(6)      // O(n)
// b = [2, 4, 4, 5, 7]

a.unique();     // O(n)
// a = [1, 2, 3, 2, 4]

a.sort();       // O(nlogn) - stable
// a = [1, 2, 2, 3, 4]

a.splice(a.end(), x, it, b.end()); // O(1)
// a = [1, 2, 2, 3, 4, 5, 7]
// b = [2, 4, 4]
```

List

Why use this instead of a Vector

```
a.merge(b);                 // O(n)
// a = [1, 2, 2, 2, 3, 4, 4, 4, 5, 7]
// b = []

a.reverse();                // O(n)
// a = [7, 5, 4, 4, 4, 3, 2, 2, 2, 1]

bool is_even(int x) { return x % 2 == 0; }
a.remove_if(is_even);       // O(n)
// a = [7, 5, 3, 1]
```

Some others

They do what you think they do.

- `#include <queue>`
- `#include <stack>`

Sorted Set

```
#include <set>

s.insert(3);    // O(logn)
s.erase(3);     // O(logn)

// Remove all integers x such that 10 <= x <= 100
s.erase(s.lower_bound(10), s.lower_bound(100))

// Find the smallest integer bigger than 9000
s.upper_bound(9000)

// Check if an element is in s. O(logn)
s.find(x) != s.end();
// or
s.count(x) != 0;
```

Sorted Set

- This is a sorted container. So you can insert things in any order, then this will output them in order:

```
for (set<int>::iterator it = s.begin();
     it != s.end(); ++it)
  cout << (*s) << endl;
```

- Elements can only be in the set once. Use multiset to have elements more than once.
- You can change what it means for elements to be the same.

```
struct same_last_digit {
  bool operator()(int x, int y) const {
    return (x % 10) < (y % 10);
  }
};
set<int, same_last_digit> s;
s.insert(101);
assert(s.count(321) != 0);
```