STL Quick Reference for CS 241

The purpose of this document is to provide basic information those elements of the C++ standard library we think are most likely to be needed in CS 241.

1 Pairs (`#include <utility>`)

The `std::pair` template provides an easy means of grouping two items into a structure:

```
std::pair<int, char> p(1, 'a');
std::cout << p.first << ' ' << p.second << std::endl; // 1 a
```

Pairs have a constructor for easy initialization of the two fields. The fields of the pair are called `first` and `second`, and they may be accessed directly (note that, in the code pictured above, these are not method calls), because `std::pair` is defined as a template struct, rather than as a class.

2 Vectors (`#include <vector>`)

The `std::vector` template provides dynamically-sized, heap-based arrays, without the need for explicit memory management:

```
std::vector<int> v;
v.push_back(2);
v.push_back(4);
v.push_back(6);
std::cout << v[0] <<' '<< v[1] <<' '<< v[2] << std::endl; // 2 4
   6
```

2.1 Constructors

There are several options for constructing vectors

- Default constructor: creates an empty vector.
- Copy constructor: creates a copy of an existing vector.
- `vector<int> v(4, -25)` Creates a vector with 4 elements, all initialized to 25.
- `vector<int> v(4)` Creates a vector with 4 elements, all default-constructed (which, for ints, means 0).

2.2 Reserving space

Although vectors will grow as needed to hold whatever data you need to store, you can optimize their behaviour by reducing the amount of reallocation and copying if you can provide a rough estimate of how much space you will need:

```
std::vector<int> v;
v.reserve(50); // Pre-allocate space for 50 items
```

2.3 Iteration

Looping through vectors can be done with index variables:

```
vector<int> v;
//...
for(int i = 0; i < v.size(); ++i) {
        // ... v[i] ...
}
```

or with iterators, which are an abstraction of pointers:

```
vector<int> v;
//...
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
        // ... *it ... (*it is the current vector element)
}
```

Iteration in reverse is done via a reverse iterator:

```
vector<int> v;
//...
for (vector<int>::reverse_iterator it = v.rbegin(); it != v.rend
   (); ++it) {
        // ... *it ... (*it is the current vector element)
}
```

The following code, however, will not compile:

```
void f(const vector<int> &v) {
        //...
        for (vector<int>::iterator it = v.begin(); it != v.end();
            ++it) {
                // ... *it ... (*it is the current vector element
                  )
        }
}
```

The reason is that `v` is a constant vector, and iterating over `v` would permit its contents to be changed, by assigning to `*it`. If you need to iterate over a const vector, use a const iterator:

```
void f(const vector<int> &v) {  // This will compile
        //...
        for (vector<int>::const_iterator it = v.begin(); it != v.
           end(); ++it) {
                // ... *it ... (*it is the current vector element
                  )
        }
}
```

2.4 Insertion and Deletion

Insertion is accomplished via the `insert` method, which takes an iterator and a value:

```
vector<int> v;
//...
v.insert(v.begin() + 3, 22); // Insert 22 at position 3 (starting
    from 0).
v.insert(v.end() - 2, 55); // Insert 55 at third-to-last position
```

Elements are deleted from the vector using the `erase` method, which can operate on either a single element or on a range of elements.

```
vector<int> v;
//...
v.erase(v.begin() + 3);  // Remove item at position 3 (starting
   from 0).
v.erase(v.begin() + 10, v.begin() + 15); // Remove items 10 ..
   15.
```

Keep in mind that insertion and deletion from positions other than the end of the vector will cause items to be shuffled, which can be inefficient.

2.5 Converting to an array

If you have a vector of items, but you need to call a function that takes an array of items, what do you do?

```
void f(int *a);
vector<int> v;
// want to call f(v);
```

Take advantage of the fact that vectors are guaranteed to be implemented as arrays and pass a pointer to the beginning of the underlying array to the function:

```
void f(int *a);
vector<int> v;
f(&v[0]);
```

3 Deques (`#include <deque>`)

The `std::deque` template provides a double-ended queue. Deques offer a similar interface to vectors, but additionally provide efficient insertion and deletion at the front of the data structure, as well as at the back. Thus, in addition to `vector`‘s `push_back` and `pop_back`, deques offer `push_front` and `pop_front`. Most vector operations are also available for deques, but be aware that deques are not guaranteed to be implemented internally as contiguous arrays.

4 Stacks (`#include <stack>`)

The `std::stack` template isnt really a separate data structure. It is an *adapter* template, which provides a specialized interface to an existing template (by default, stacks are built from deques, but you can specify that you want a vector- or list-based stack). Stacks provide only the basic stack operations:

```
stack<int> s;
if (!s.empty()) {  // Test for empty
        std::cout << s.size() << std::endl;  // Size of the stack
}
s.push(4);  // "push" operation
std::cout << s.top() << std::endl;  // top of stack
s.pop();  // "pop" operation
```

Note that the `pop` method does not return the item that was popped. If you want this item, use the `top` method before the `pop` method.