STL Quick Reference for CS 241

Spring 2018

The purpose of this document is to provide basic information on those elements of the C++14 standard library we think are most likely to be needed in CS 241.

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`.

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.emplace_back(2);
v.emplace_back(4);
v.emplace_back(1);
std::cout << v[0] << ' ' << v[1] << ' ' << v[2] << std::endl; // 2 4 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).
- `vector<int> v{2, 4, 1}` creates a vector holding the elements 2, 4, 1.

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
```

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)
}
```

or using a range-based loop:

```
vector<int> v;
//...
for (auto &x : v) {
  // ... x ... (x 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)
   }
}
```

or using a range-based loop:

```
void f(const vector<int> &v) {  // This will compile
   //...
   for (const auto &x : v) {
     // ... x ... (x is the current vector element)
   }
}
```

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 second-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);  // Erase 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.

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]);
```

Maps (`#include <map>`)

The `std::map` template provides a dictionary with efficient lookups. The implementation of a map is typically as a balanced binary search tree, and lookups are guaranteed to run in $O(\log n)$ time.

```
map<string,int> m;
m["abc"] = 13;
m["def"] = 57;
std::cout << m["abc"] << ' ' << m["def"] << ' ' << m["ghi"] << std::endl; // 13 57 0
```

The first type in the template instantiation is the key type, and the second type is the value type. Keys are unique within the map. With maps, you use `operator[]` to store and retrieve elements from the map, just as you would with arrays. If you attempt to reference a key that is not present in the map, an entry for the key is created, and its corresponding value is default-constructed (which, for ints, means 0). This is why the code above printed 0 at the end.

For this reason, you cannot use `operator[]` to tell whether a key is actually present in the map. For this, use the `count` method:

```
if (m.count("jkl")) { /* Returns the number of times key "jkl" occurs in m */ }
```

Since keys can occur at most once, `count` can only return either 1 or 0, corresponding to the key being, respectively, present and not present in the map.

Iteration

Iteration over maps is done with iterators, and is analogous to the case for vectors:

```
map<string,int> m;
//...
for (map<string,int>::iterator it = m.begin(); it != m.end(); ++it) {
  std::cout << it->first << ": " it->second << std::endl;
}
```

Two points are worth noting:

- The iterator (`it` above) points to a `pair<string,int>`. So to fetch the key, the correct expression to use is `it->first` (equivalently, `(*it).first`). Simalarly, `it->second` retrieves the value.
- Iteration produces (key,value) pairs in sorted key order. The order in which the items were inserted is not relevant.

We can also do iteration using a range-based loop:

```
map<string,int> m;
//...
for (auto &p : m) {
  std::cout << p.first << ": " p.second << std::endl;
}
```

Removal

Removing elements from a map is done via the `erase` method:

```
m.erase("abc");
```

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 `emplace_back` and `pop_back`, deques offer `emplace_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.

Stacks (`#include <stack>`)

The `std::stack` template isn't 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.

Unique Pointers (`#include <memory>`)

Programming in C++ becomes much easier if you take advantage of RAII classes like `vector` and `unique_ptr` to manage your memory for you, via their destructors:

```
class C {
 public:
  C(int m, int n) { ... }
};

void f() {
  unique_ptr<C> p{new C(1, 2)};  // creates a C object on the heap
  auto q = make_unique<C>(1, 2);  // also creates a C object on the heap

  // ...
  // p and q automatically cleaned up on exit; the associated heap objects
  // are deleted
}
```

The two lines above perform the same task; in general, you should prefer to use `make_unique` as it saves you from creating the heap object yourself (you don't call `delete`, so why should you call `new`?).

Unique pointers can be dereferenced via the `*` and `->` operators, just like ordinary pointers. Unique pointers cannot be copied, as that would mean two smart pointers would be taking ownership of the same resource. However, unique pointers can be moved, meaning that one unique pointer would take over the resource from the other.

Other operations on unique pointers:

```
auto raw = p.get();  // returns the underlying raw pointer, which can be passed
                     // to functions expecting non-owning pointers
p.reset(ptr);        // deletes the pointer p currently holds, and makes p
                     // hold ptr instead
p.release();         // drops the pointer p currently holds, without deleting it
```