Week 4 Notes

STL: Standard Template Library

Has three fundamental pillars:

- Containers
  - ex. stack, list, array, etc.
- Iterators
  - ex. input_iterator, forward_iterator, bidirectional_iterator, etc.
- Algorithms
  - ex. sort, find, binary_search, count, etc.

They are all fully generic, can work with anything

Generics

- Can easily change the types stored
  - There's only actually one implementation of stack under the hood
- For initializing generics, use `container<type>`
- Can use any user-defined type as well
- Using generics in functions and data types allows us to write one function that works with many types

Vectors

- Can define similar to stacks: `vector<T>`

Vector provides:

- A sequence, similar to a C array (except for added safety)
- What makes them stand out, though, is that they are *dynamically sized*
  - Similar to list in Python
  - Vectors grow to accomodate elements added
- Has ability to add/remove anywhere
- Can be indexed like an array
- Has bounds safety

Vector Usage Example

```
#include <iostream>
#include <vector>

using std::cout, std::endl, std::vector;

int sum(vector<int>& vec) { // Pass by reference
    cout << "In sum(vector<int>& vec)"
    int total = 0;
    for (int x: vec) {
        total += x;
    }
    return total;
}

double sum(vector<double>& vec) {
    cout << "In sum(vector<double>& vec)"
    double total = 0;
    for (double x: vec) {
```

```
        total += x;
    }
    return total;
}

int main() {
    // Replace all int types in main with double to test function overloading
    // Overloading like this is bad practice, because it has a lot of repeat code
    // This can be fixed using templates in the next example
    vector<int> vec;

    for (int i = 0; i < 10; i++) {
        vec.push_back(i) // Similar to list.append in Python
    }

    vec[0] = 1000;

    for (int x: vec) { // Fancy for loop formatting
        cout << x << " "
    }
    cout << endl;

    cout << sum(vec) << endl;

    return 0;
}
```

Templates Example

```
#include <iostream>
#include <vector>
#include <string>

using std::cout, std::endl, std::vector, std::string;

template <class T>
T sum(vector<T>& vec) { // Returning generic type T
    T total{}; // Initializing object of type T
    for (T x: vec) {
        total += x; // This will work for any type that has the addition operation defined
    }
    return total;
}

int main() {
    vector<string> vec;

    vec.push_back(string("Hello"))
    vec.push_back(string("World!"))

    cout << sum(vec) << endl;

    return 0;
}
```

Iterators

- Iterators are abstractions for indexes into containers
  - Iterators allow you to point to indexes in a structure without knowing how it's represented
    - ex: Can iterate over a binary tree
- Can make iterator point to beginning or make it point past the end
- Can move iterator forward, backward, or to a specific rank
- Iterators are used under the hood in loops

Ranks

In an array, ranks == index - For other data types, we use ranks to talk about the i'th element

STL and Iterators

- The container has methods to manufacture iterators
  - At either end, forward or reversed
- Iterators point to an index of a container
- Iterators can be sort-of compared to pointers

Iterator Example

```
#include <iostream>
#include <vector>

using std::cout, std::endl, std::vector;

int main() {
    vector<int> vec;

    for (int i = 0; i < 5; i++) { // Loops use iterators under the hood
        vec.push_back(i);
    }

    // Forward iteration
    for (
        vector<int>::iterator it = vec.begin(); // int i = 0;
        it != vec.end(); // i < vec.size();
        it++ // i++;
    ) {
        cout << *it << endl;
    }

    // Reverse iteration
    for (
        vector<int>::reverse_iterator it = vec.rbegin(); // int i = 0;
        it != vec.rend(); // i < vec.size();
        it++ // i++;
    ) {
        cout << *it << endl;
    }

    return 0;
}
```

Type Inferencing

- `auto` keyword
- Useful for complex types
- Can use anywhere, but you shouldn't
  - If it makes it more readable, use `auto`; if not, don't use it
- Cannot use `auto` for argument types
- Can also use references in range loops
  - `for (const auto &x: vec)` over `for (T x: vec)`

Auto Example

```cpp
#include <iostream>
#include <vector>

using std::cout, std::endl, std::vector;

template <class Container>
void rprint(Container& con) {
    for (auto it = con.rbegin(); it != con.rend(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}

int main() {
    vector<int> vec;
    for (int i = 0; i < 10; i++) {
        vec.push_back(i);
    }
    rprint(vec);

    return 0;
}
```