CSE 333 Section 6

Templates and STL

UNIVERSITY of WASHINGTON

Logistics

- Midterm
  - Released **Wednesday** (2/09) @ 12:00am
  - Due **Saturday** (2/12) @ 11:59pm
- Exercise 8
  - Released **Wednesday** (2/09)
  - Due **Wednesday** (2/16) @ 11am

Templates!

Templates

- C++ syntax to generate code that works with *generic types*
- Generates a new implementation in assembly for every type it is used with:
  - *e.g.*, calls to `foo<int>()` and `foo<double>()` generate two implementations
  - *e.g.*, calls to `foo<int>()` and another `foo<int>()` require only one implementation
  - *e.g.*, if `foo` is never used, zero implementations are generated

Template Function

```
template<typename T>
T add3(T arg) {
    T result = arg + 3;
    return result;
}
```

Results?

```
add3<int>(3);           // uses add3<int>, returns 6
add3(5.5);
add3<char*>("a str");
add3<string>("a str");
```

Template Function

```
template<typename T>
T add3(T arg) {
    T result = arg + 3;
    return result;
}
```

Results?

```
add3<int>(3);           // uses add3<int>, returns 6
add3(5.5);              // uses add3<double>, returns 8.5
add3<char*>("a str");
add3<string>("a str");
```

Template Function

```
template<typename T>
T add3(T arg) {
    T result = arg + 3;
    return result;
}
```

Results?

```
add3<int>(3);           // uses add3<int>, returns 6
add3(5.5);              // uses add3<double>, returns 8.5
add3<char*>("a str");   // uses add3<char*>, return ->"tr"
add3<string>("a str");
```

Template Function

```
template<typename T>
T add3(T arg) {
    T result = arg + 3;
    return result;
}
```

Results?

```
add3<int>(3);           // uses add3<int>, returns 6
add3(5.5);              // uses add3<double>, returns 8.5
add3<char*>("a str");   // uses add3<char*>, return ->"tr"
add3<string>("a str"); // Compiler error! No `+` for string
                        //                 and int
```

Template Function

```
template<typename T>
T add3(T arg) {
    T result = arg + 3;
    return result;
}
```

- C++ templates are “duck typed” in the version of C++ we use in this class
- If there’s a way for the compiler to substitute a type, it will, even if the results are somewhat unexpected, such as the `char*` substitution

Template Function

```
template<typename T, int N = 2> // Templatize values
T modulo(T arg) {
    T result = arg % N;
    return result;
}

modulo(5) == 1  (=5%2)
modulo<int, 5>(17) == 2  (=17%5)

// C++ template system is very powerful
// Simple type-substitution is enough for most programs
```

Template Class

- Very useful for implementing data structures that support *generic types*:

```
typedef uint64_t HTKey_t;
typedef void*    HTValue_t;
typedef struct {
  HTKey_t   key;
  HTValue_t value;
} HTKeyValue_t;
```

```
template<typename K, typename V>
struct HTKeyValue {
  K HTKey;
  V* HTValue;
};
```

Exercise 1

Exercise 1

```
_______________                  // template type definition
struct Node {
  _______________                // two-argument constructor

  ~Node() { delete value; }      // destructor cleans up the payload

  _______________                // public field value
  _______________                // public field next
};
```

Exercise 1 Solution

```
template <typename T>          // template type definition
struct Node {
  ______________              // two-argument constructor

  ~Node() { delete value; }   // destructor cleans up the payload

  ______________              // public field value
  ______________              // public field next
};
```

Exercise 1 Solution

```
template <typename T>          // template type definition
struct Node {
  ______________              // two-argument constructor

  ~Node() { delete value; }   // destructor cleans up the payload

  T* value                    // public field value
  Node<T>* next               // public field next
};
```

Exercise 1 Solution

```
template <typename T>          // template type definition
struct Node {
  Node(T* val, Node<T>* node): value(val), next(node) {}
                              // two-argument constructor

  ~Node() { delete value; }   // destructor cleans up the payload

  T* value                    // public field value
  Node<T>* next               // public field next
};
```

Containers!

C++ standard lib is built around templates

- ***Containers*** store data using various underlying data structures
  - The specifics of the data structures define properties and operations for the container
- ***Iterators*** allow you to traverse container data
  - Iterators form the common interface to containers
  - Different flavors based on underlying data structure
- ***Algorithms*** perform common, useful operations on containers
  - Use the common interface of iterators, but different algorithms require different ‘complexities’ of iterators

Common C++ STL Containers (and Java equiv)

- *Sequence* containers can be accessed sequentially
  - `vector<Item>` uses a dynamically-sized contiguous array (like `ArrayList`)
  - `list<Item>` uses a doubly-linked list (like `LinkedList`)
- *Associative* containers use search trees and are sorted by keys
  - `set<Key>` only stores keys (like `TreeSet`)
  - `map<Key,Value>` stores key-value `pair<>`'s (like `TreeMap`)
- *Unordered associative* containers are hashed
  - `unordered_map<Key,Value>` (like `HashMap`)

Common C++ STL Methods

	vector	list	set	map	unordered_map
`.size()` // get number of elements	✔	✔	✔	✔	✔
`.push_back()` // add element to back `.pop_back()` // remove back element	✔	✔			
`.push_front()` // add element to front `.pop_front()` // remove front element		✔			
`.operator[]()` // random access element	✔			✔	✔
`.insert()` // insert key			✔	✔	✔
`.find()` // find key			✔	✔	✔

Common STL Containers

Many more containers and methods!

See full documentation here:
http://www.cplusplus.com/reference/stl

Exercise 2

Exercise 2

```
using namespace std;
vector<string> ChangeWords(const vector<string>& words,
                           map<string,string>& subs) {









}
```

Exercise 2 Solution

```
using namespace std;
vector<string> ChangeWords(const vector<string>& words,
                           map<string,string>& subs) {
  vector<string> result;
  for (auto& word : words) {
    if (subs.find(word) != subs.end()) {
      result.push_back(subs[word]);
    } else {
      result.push_back(word);
    }
  }
  return result;
}
```

Exercise T9

Exercise T9 Set up

Before smartphones, mobile phones used a predictive text system called T9, based on the mapping of a single numpad key to any of the corresponding letters shown in the image to the right. Note that the ‘1’, ‘*’, and ‘#’ keys won’t be used and that ‘0’ corresponds to [Space].

Example: a user would type ‘8’, then ‘4’, then ‘3’ to get the word “the”, though it could also predict longer words like “they” or “there”.

We will use C++ STL to generate our T9 predictive dictionary!

Exercise T9 A

```
map<string, vector<string>> predictions; // global prediction map
void AddPrefixesToPredictions(const string& word) {









}
```

Exercise T9 A Solution

```
map<string, vector<string>> predictions; // global prediction map
void AddPrefixesToPredictions(const string& word) {

 string prefix;

 for (auto& c : word) {
   prefix += letters_to_keys[c];
   predictions[prefix].push_back(word);
 }



}
```

Exercise T9 A Solution 2

```
map<string, vector<string>> predictions; // global prediction map
void AddPrefixesToPredictions(const string& word) {
 // soln 2: extra loop to push *word to onto all prefix keys

 for (auto c : word) {
   prefix += letters_to_keys[c];
 }
 for (size_t i = 1; i <= prefix.length(); i++) {
   predictions[prefix.substr(0,i)].push_back(word);
 }

}
```

Exercise T9 B

```
map<string, vector<string>> predictions; // global prediction map
void PrintPredictions() {









}
```

Exercise T9 B Solution

```
map<string, vector<string>> predictions; // global prediction map
void PrintPredictions() {
 // loop over every prediction pair
 for (auto& pred_pair : predictions) {
   cout << pred_pair.first << " : ";

   // loop over every vector entry
   for (auto& w : pred_pair.second) {
     cout << w << ", ";
   }
   cout << endl;
 }
}
```

Thanks for coming to section!

Reminder!

Midterm	Released Wednesday 12am, due Saturday 11:59pm
Exercise 8	Due Wednesday 11am