CSE 333 Section 6

Templates, STL

UNIVERSITY of WASHINGTON

Logistics

- Midterm
  - Released today
  - No lecture tomorrow (02/10)
  - Due **Saturday** (2/11) @ 11:59pm
- Exercise 8
  - Released **Wednesday** (2/08)
  - Due **Wednesday** (2/15) @ 11am

Templates!

C++ 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

C++ Template Function

```
template<typename T>
T Add3(T arg) {
    T result = arg + 3;
    return result;
}
```

What is the result of each line of code?

```
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
```

C++ Template Class

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

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

C

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

C++

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
};
```

C++

Exercise 1

```
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
};
```

C++

Exercise 1

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

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

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

C++

Exercise 1

```
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
};
```

C++

Exercise 1

```
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
};
```

C++

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	✔			✔	✔
`.find()` // find key			✔	✔	✔

Common STL Containers

Many more containers and methods!

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

Helpful C++ Features

- Using `auto` as a data type asks the compiler to infer for you – can save you a lot of typing, but makes it easier to lose track of types
  - Can add & (*e.g.*, `auto& var`) to assign by reference instead of copying
- Range-for statement
  - Similar to Java's `foreach`, with *`decl`* defining the loop variable and *`expr`* being the sequence to loop over

```
for ( decl : expr ) {
  statements
}
```

Exercise 2

Exercise 2

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









}
```

Exercise 2

```
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”.
Online T9 Emulator: https://www.sainsmograf.com/labs/t9-emulator/

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

Exercise T9 Set up

prefix		**Predicted word list** (red words are what user get)
843	→	the tid … they there these …
8439	→	they …
84373	→	there these …

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”.
Online T9 Emulator: https://www.sainsmograf.com/labs/t9-emulator/

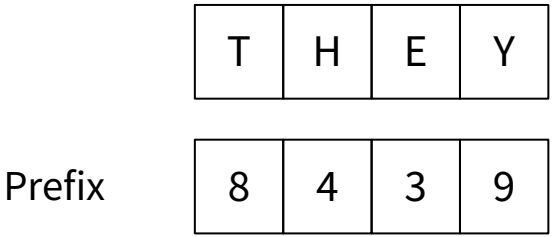
We will use C++ STL to generate our T9 predictive dictionary!

Exercise T9 A

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









}
```

Exercise T9 A

```
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);
 }



}
```

T	H	E	Y

Prefix

8	4	3	9

Exercise T9 B

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



}
```

```
2 : a, ax,
29 : ax,
```

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 << " : ";
   …
   …


 }
}
```

```
2 : a, ax,
29 : ax,
```

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;
 }
}
```

```
2 : a, ax,
29 : ax,
```

Thanks for coming to section!