333 Section 6 - C++ Templates, STL

Welcome back to Section! We’re glad that you’re here :) 

**C++ Templates**

An example converting an existing function to use templates is below (notice that in the template version `N` is also passed in via template parameter whereas in the regular version it is a parameter):

```cpp
Non-Template: 
int Modulo(int arg, int n) {
    int result = arg % n;
    return result;
}

Template:  
template<typename T, int N = 2>
T Modulo(T arg) {
    T result = arg % N;
    return result;
}
```

Templates can also be used for classes. A member variable of a template class can be declared using one of the class’ template types. This is very useful for implementing data structures that support generic types:

**Generic HTKeyValue using C++ template:**

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

**Generic HTKeyValue_t from HW1:**

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

On the right is the `HTKeyValue_t` struct definition from HW1, look how much cleaner it is using C++ template!

**Exercise 1) Template Class**

Fill in the blanks below for the definition of a simple templated struct `Node` for a singly-linked list. The struct has two public fields: a `value`, which is a pointer of template type `T` pointing to a heap allocated payload, and a `next`, which is a pointer to another struct `Node`. The struct also has a two-argument constructor that takes a `T` pointer for `value` and another `Node<T>` pointer for `next`.

```cpp
struct Node {
    // template type definition
    __________
    // two-argument constructor
    __________
    ~Node() { delete value; } // destructor cleans up the payload
    __________
    // public field value
    __________
    // public field next
};
```
C++’s Standard Template Library (STL)
Containers, iterators, algorithms (sort, find, etc.), numerics
- **general** - .begin(), .end(), .size(), .erase()
- **template <class T> class std::vectors** - .operator[](), .push_back(), .pop_back()
- **template <class T> class std::list** - .push_back(), .pop_back(), .push_front(), .pop_front(), .sort()
- **template <class Key, class T> class std::map** - .operator[](), .insert(), .find(), .count()
- **template <class T1, class T2> struct std::pair** - .first, .second

Exercise 2) Standard Template Library
Complete the function ChangeWords below. This function has as inputs a vector of strings, and a map of <string, string> key-value pairs. The function should return a new vector<string> value (not a pointer) that is a copy of the original vector except that every string in the original vector that is found as a key in the map should be replaced by the corresponding value from that key-value pair.

Example: if vector words is {"the", "secret", "number", "is", "xlii"} and map subs is {{"secret", "magic"}, {"xlii", "42"}}, then ChangeWords(words, subs) should return a new vector {"the", "magic", "number", "is", "42"}.

**Hint:** Remember that if m is a map, then referencing m[k] will insert a new key-value pair into the map if k is not already a key in the map. You need to be sure your code doesn’t alter the map by adding any new key-value pairs. (Technical nit: subs is not a const parameter because you might want to use its operator[] in your solution, and [] is not a const function. It’s fine to use [] as long as you don’t actually change the contents of the map subs.)

Write your code below. Assume that all necessary headers have already been written for you.

```cpp
using namespace std;

vector<string> ChangeWords(const vector<string>& words, map<string, string>& subs) {
    // Your code here
}
```
Exercise 3) STL Debugging [extra]
Here is a little program that has a small class Thing and main function (assume that necessary
#include and using namespace std; are included).

```cpp
class Thing {
  public:
    Thing(int n): n_(n) { }
    int getThing() const { return n_; }
    void setThing(int n) { n_ = n; }
  private:
    int n_;  
  };

int main() {
  Thing t(17);
  vector<Thing> v;
  v.push_back(t);
}
```

This code compiled and worked as expected, but then we added the following two lines of code (plus
the appropriate #include <set>):

```cpp
set<Thing> s;
s.insert(t);
```

The second line (s.insert(t)) failed to compile and produced dozens of spectacular compiler error
messages, all of which looked more-or-less like this (edited to save space):

```
In file included from string:48:0, from bits/locale_classes.h:40, from
bits/ios_base.h:41, from ios:42, from ostream:38, from /iostream:39, from
thing.cc:3: bits/stl_function.h: In instantiation of 'bool
std::less<_Tp>::operator()(const _Tp&, const _Tp&) const [with _Tp =
Thing]': <<many similar lines omitted>> thing.cc:37:13: required from here
bits/stl_function.h:
387:20: error: no match for 'operator<' (operand types are 'const Thing'
and 'const Thing') { return __x < __y; } 
```

What on earth is wrong? Somehow class Thing doesn't work with set<Thing> even though insert
is the correct function to use here. (a) What is the most likely reason, and (b) what would be needed
to fix the problem? (Be brief but precise – you don’t need to write code in your answer, but you can if
that helps make your explanation clear.)
T9 Example
Before smart phones, 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!

The top of our file is shown below so that you are aware of what is globally available:

```
#include <iostream>
#include <string>
#include <vector>
#include <map>
using namespace std;
```

Our T9 class also has a field `map<char, char> letters_to_keys`, which maps letters to their corresponding number on the T9 keyboard. For this exercise, assume this map has already been initialized for you.

a) Complete the function to add a mapping from each prefix to the string itself to predictions. Assume the passed-in word is always lowercase. You may find the string member function `string substr(size_t pos, size_t len)` useful, which returns the substring of length `len` starting from position `pos`.

```
map<string, vector<string>> predictions; // global prediction map.
void AddPrefixesToPredictions(const string& word) {
    ```
b) Complete the function below to print out the contents of \texttt{predictions}. For example, if we've added "a" and "ax", it should print out the following (note the formatting):

\begin{verbatim}
  2 : a, ax,
  29 : ax,
\end{verbatim}

```c
void PrintPredictions() {
}
```