5950 Section 8 - Copy Constructor, Templates, and STL

Welcome back to recitation! We're glad that you're here :)

Copy Constructor

In C++ we can construct objects based on an already existing object with the copy constructor. C++ will synthesize one for us that works well with "simple" objects, but we encounter issues if there is any dynamically allocated memory, or resources acquired (like an open file). Consider the following example:

Exercise:

1) Bad Copy

Identify the memory error with the following code.

```
class BadCopy {
 public:
  BadCopy()  { arr_ = new int[5]; }
  ~BadCopy() { delete [] arr_; }
 private:
  int *arr_;
};

int main(int argc, char **argv) {
  BadCopy *bc1 = new BadCopy;
  BadCopy *bc2 = new BadCopy(*bc1);  // BadCopy's cctor

  delete bc1;
  delete bc2;

  return EXIT_SUCCESS;
}
```

The default copy constructor does a shallow copy of the fields, so `bc2`'s `arr_` points to the same array as `bc1`'s `arr_`. When `bc1` gets deleted, so does its `arr_`. But this `arr_` is the same one `bc2`'s `arr_` points to, so when `bc2` gets deleted, its `arr_` has already been deleted, leading to an invalid delete (similar to a double `free()`).

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):

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:

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

Generic `HTKeyValue_t` in C:

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

On the right is an `HTKeyValue_t` struct definition for defining a key-value pair for a HashTable in C, look how much cleaner it is using C++ template!

Exercise:

2) 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`.

```
template <typename T>
struct Node {
  Node(T* val, Node<T>* node): value(val), next(node) {}

  ~Node() { delete value; }

  T* value;
  Node<T>* next;
};
```

Remember that struct in C++ by default has its members being public, so no need to specify the access modifiers explicitly here

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`

Exercises:

3) 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 `operator[]` does not work on a const map. 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.

```
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;

  }
```

4) STL Debugging

Here is a little program that has a small class `Thing` and main function (assume that necessary `#includes` and `using namespace std;` are included).

```
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>`):

```
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.)

STL has to compare them using operator<. Add an appropriate operator< as either a member function in Thing, or as a free-standing function that compares two Thing& parameters.