

333 Section 6 - C++ Templates, STL, and Smart Pointers

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):

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` from HW1:

```
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`.

```
_____ // template type definition
struct Node {
    _____ // 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::vector** – `.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:

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.

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

```
}
```

3) STL Debugging [Extra exercise]

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.)

C++ Smart Pointers

C++'s smart pointers can be used to automatically manage memory if used properly.

- `std::unique_ptr` – `.get()`, `.release()`, `.reset()`
- `std::shared_ptr` – `.get()`, `.use_count()`, `.unique()`
- `std::weak_ptr` – `.lock()`, `.use_count()`, `.expired()`

4) “Smart” LinkedList

Consider the `Node` struct that you completed in Exercise 1 below. Convert the `Node` struct to be “smart” by using `shared_ptr`s.

```
#include <memory>
using std::shared_ptr;

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

    ~Node() { delete value; }

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

After the conversion, we should be able to get rid of the destructor and the following program that uses this `Node` struct should have no memory leak. (Note that we never called `delete` ourselves!)

```
#include <iostream>

using std::cout;
using std::endl;

int main() {
    shared_ptr<Node<int>> head =
        shared_ptr<Node<int>>(new Node<int>(new int(351),
            nullptr));
    head->next = shared_ptr<Node<int>>(new Node<int>(new int(333), nullptr));
    shared_ptr<Node<int>> iter = head;
    while (iter != nullptr) {
        cout << *(iter->value) << endl;
        iter = iter->next;
    }
}
```