CSE 333 Section 5 - Heap, Templates, STL

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

Exercise 1

```
#include <cstdlib>

class HeapInt{
 public:
  HeapInt() { x_ = new int(5); }
  ~HeapInt () { delete x_; }  // Delete the allocated int
 private:
  int* x_;
};

int main(int argc, char** argv) {
  HeapInt** hpint_ptr = new HeapInt*;
  HeapInt* hpint = new HeapInt();
  *hpint_ptr = hpint;
  delete hpint_ptr;
  delete hpint;  // Delete of hpint_ptr doesn't delete what hpint
points to
  return EXIT_SUCCESS;
}
```

Assuming an instance of `HeapInt` takes up 8 bytes (like a C-struct with just `int* x_`), how many bytes of memory are leaked by this program? How would you fix the memory leaks?

Leaks 12 bytes of memory: 8 bytes for the allocated HeapInt object `hpint` points to + 4 bytes for the `int` the HeapInt instance allocates in its constructor.

Deleting the `hpint_ptr` doesn't automatically delete what the pointer points to. Have to also delete `hpint` and then create a destructor that deletes the allocated int pointer `x_`.

Exercise 2

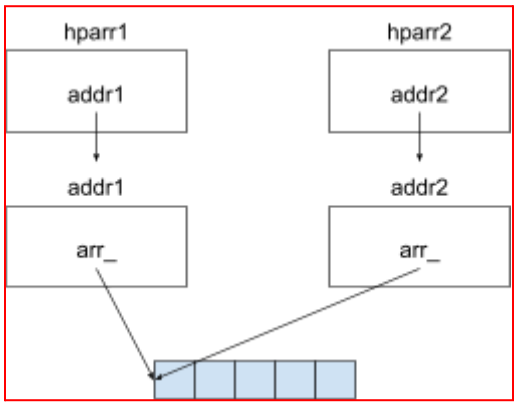
```
class HeapArr{
 public:
  HeapArr()  { arr_ = new int[5]; }
  ~HeapArr() { delete [] arr_; }
 private:
  int* arr_;
};

int main(int argc, char** argv) {
  HeapArr* hparr1 = new HeapArr;
  HeapArr* hparr2 = new HeapArr(*hparr1);  // HeapArr's cctor

  delete hparr1;
  delete hparr2;

  return EXIT_SUCCESS;
}
```

Identify the memory error with the following code. Then fix it! Hint: Draw a memory diagram. What happens when `hparr1` gets deleted?



The default copy constructor does a shallow copy of the fields, so `hparr2`'s `arr_` points to the same array as `hparr1`'s `arr_`. When `hparr1` gets deleted, so does its `arr_`. But this `arr_` is the same one `hparr2`'s `arr_` points to, so when `hparr2` gets deleted, its `arr_` has already been deleted, leading to an invalid delete (similar to a double `free()`).

C++ Templates

Exercise 3) Templates & Things

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 Library

Exercise 4) 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) {

  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;

}
```