Section Handout 5

Problem One: Constructors and Destructors

Constructors and destructors are unusual functions in that they're called automatically in many contexts and usually aren't written explicitly. To help build an intuition for when constructors and destructors are called, trace through the execution of this program and list all times when a constructor or destructor are called.

```
/* Prints the elements of a stack from the bottom
 * of the stack up to the top of the stack.  To
 * do this, we transfer the elements from the
 * stack to a second stack (reversing the order
 * of the elements), then print out the contents
 * of that stack.
 */
void printStack(Stack<int> toPrint) {
      Stack<int> temp;
      while (!toPrint.isEmpty())
         temp.push(toPrint.pop());

      while (!temp.isEmpty())
         cout << temp.pop() << endl;
}

int main() {
      Stack<int> elems;
      for (int i = 0; i < 10; i++)
         elems.push(i);

      printStack(elems);
}
```

Problem Two: Writing Destructors

In lecture, we wrote an `IntStack` class that represents a stack of `int`s. The private data members of the class were

```
private:
    int* elems;
    int logicalLength;
    int allocatedLength;
```

The destructor for the class was defined as follows:

```
IntStack::~IntStack() {
    delete[] elems;
}
```

Why doesn't the destructor do anything to `logicalLength` or `allocatedLength`? Wouldn't it make sense to set them to 0?

Problem Three: Pointer Traces!

Pointers to arrays are different in many ways from **Vector** or **Map** in how they interact with pass-by-value and the **=** operator. To better understand how they work, trace through the following program. What is its output?

```
#include <iostream>
using namespace std;

void print(int* first, int* second) {
      for (int i = 0; i < 5; i++) {
            cout << i << ": " << first[i] << ", " << second[i] << endl;
      }
}

void transmogrify(int* first, int* second) {
      for (int i = 0; i < 5; i++) {
            first[i] = 137;
      }
}

void mutate(int* first, int* second) {
      first = second;
      second[0] = first[0];
}

void change(int* first, int* second) {
      first = new int[5];
      second = new int[5];

      for (int i = 0; i < 5; i++) {
            first[i] = second[i] = 271;
      }
}

int main() {
      int* one = new int[5];
      int* two = new int[5];

      for (int i = 0; i < 5; i++) {
            one[i] = i;
            two[i] = 10 * i;
      }

      transmogrify(one, two);
      print(one, two);

      mutate(one, two);
      print(one, two);

      change(one, two);
      print(one, two);

      delete[] one;
      delete[] two;
}
```

Problem Four: **`new[]`** and **`delete[]`**

Whenever you allocate an array with **`new[]`**, you need to deallocate it using **`delete[]`**. It's important when you do so that you only deallocate the array exactly once – deallocating an array zero times causes a memory leak, and deallocating an array multiple times usually causes the program to crash.

Below are three code snippets. Trace through each snippet and determine whether all memory allocated with **`new[]`** is correctly deallocated exactly once. If there are any other errors in the program, make sure to report them as well.

```
int main() {
 int* baratheon = new int[3];
 int* targaryen = new int[5];

 baratheon = targaryen;
 targaryen = baratheon;

 delete[] baratheon;
 delete[] targareon;
}
```

```
int main() {
 int* stark     = new int[6];
 int* lannister = new int[3];

 delete[] stark;
 stark = lannister;

 delete[] stark;
}
```

```
int main() {
 int* tyrell = new int[137];
 int* arryn  = tyrell;

 delete[] tyrell;
 delete[] arryn;
}
```