Section Handout 5

Problem One: Pointed Points about Pointers

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?

```
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;
      return 0;
}
```

Problem Two: Cleaning Up Your Messes

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. (Fun fact – deallocating memory twice is called a *double free* and can lead to security vulnerabilities in your code! Take CS155 for details.)

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;

 return 0;
}
```

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

 delete[] stark;
 stark = lannister;

 delete[] stark;

 return 0;
}
```

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

 delete[] tyrell;
 delete[] arryn;

 return 0;
}
```

Problem Three: Creative Destruction

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);
      return 0;
}
```

Problem Four: Random Bag Grab Bag

The very first container class we implemented was the ***random bag***, which supported two operations:

- `add`, which adds an element to the random bag, and
- `removeRandom`, which chooses a random element from the bag and returns it.

Below is the code for the `RandomBag` class. First, `RandomBag.h`:

```
#ifndef RandomBag_Included
#define RandomBag_Included

#include "vector.h"

class RandomBag {
public:
    void add(int value);
    int removeRandom();

    int size() const;
    bool isEmpty() const;

private:
    Vector<int> elems;
};

#endif
```

Next, `RandomBag.cpp`:

```
#include "RandomBag.h"
#include "random.h"

int RandomBag::size() const {
    return elems.size();
}

bool RandomBag::isEmpty() const {
    return size() == 0;
}

void RandomBag::add(int value) {
    elems += value;
}

int RandomBag::removeRandom() {
    if (isEmpty()) {
        error("That which is not cannot be!");
    }

    int index = randomInteger(0, size() - 1);
    int result = elems[index];

    elems.remove(index);
    return result;
}
```

Let's begin by reviewing some aspects of this code.

i. What do the `public` and `private` keywords mean in `RandomBag.h`?

ii. What does the `::` notation mean in C++?

iii. What does the `const` keyword that appears in the declarations of the `RandomBag::size()` and `RandomBag::isEmpty()` member functions mean?

Now that you're a bit more acclimated to the syntax, let's look at this code in a bit more detail.

Internally, the `Vector` is implemented very much in the same way as our `IntStack` – it has a dynamic array of elements that doubles whenever the size would exceed the capacity. This means that the `+=` operator on a `Vector` runs in *amortized* time O(1), the same as our `IntStack`'s push function.

The square brackets on the `Vector` type select a single element out of its underlying array, which takes time O(1) – remember that accessing any element of a dynamically-allocated array takes the same amount of time, namely time O(1).

The `Vector`'s `.remove()` function, on the other hand, takes time proportional to the distance between the element being removed and the last element of the `Vector`. The reason for this is that whenever a gap opens up in the `Vector`, it has to shift all the elements after the gap down one spot. As a result, removing from the very end of a `Vector` is quite fast – it runs in time O(1) – but removing from the very front of a `Vector` takes time O(n).

iv. Look at the implementation of our `RandomBag::removeRandom` function. What is its worst-case time complexity? How about its best-case time complexity? Its *average*-case time complexity?

v. Based on your answer to the previous part of this question, what is the worst-case time complexity of removing all the elements of an n-element `RandomBag`? What's the best-case time complexity? How about its average case?

vi. In the preceding discussion, we mentioned that removing the very last element of a `Vector` is much more efficient that removing an element closer to the middle. Rewrite the member function `RandomBag::removeRandom` so that it always runs in worst-case O(1) time.

vii. The `Stack` and `Queue` types each have `peek` member functions that let you see what element would be removed next without actually removing anything. How might you write a member function `RandomBag::peek` that works in the same way? Make sure that the answer you give back is actually consistent with what gets removed next and that calling the member function multiple times without any intervening additions always gives the same answer.

Problem Five: Stackity Stack

In class, we implemented the `OurStack` class, which represents a stack of integers. The class definition is shown here:

```
class OurStack {
public:
    OurStack();
    ~OurStack();

    void push(int value);
    int  pop(int value);
    int  peek(int value) const;

    int  size() const;
    bool isEmpty() const;

private:
    void grow();

    int* elems;
    int logicalSize;
    int allocatedSize;
};
```

The implementation is on the next page, just as a refresher.

Here's the .cpp file:

```
#include "OurStack.h"
#include "error.h"

const int kInitialSize = 4;

OurStack::OurStack() {
    elems = new int[kInitialSize];
    logicalSize = 0;
    allocatedSize = kInitialSize;
}

OurStack::~OurStack() {
    delete[] elems;
}

void OurStack::push(int value) {
    if (logicalSize == allocatedSize) {
        grow();
    }

    elems[logicalSize] = value;
    logicalSize++;
}

int OurStack::peek() const {
    if (isEmpty()) {
        error("What is the sound of one hand clapping?");
    }
    return elems[logicalSize - 1];
}

void OurStack::pop() {
    int result = peek();
    logicalSize--;
    return result;
}

int OurStack::size() const {
    return logicalSize;
}

bool OurStack::isEmpty() const {
    return size() == 0;
}

void OurStack::grow() {
    int* newArr = new int[2 * allocatedSize];
    for (int i = 0; i < size(); i++) {
        newArr[i] = elems[i];
    }

    delete[] elems;
    elems = newArr;
    allocatedSize *= 2;
}
```

This problem consists of some questions about the `OurStack` type:

i. What is the meaning of the term "logical size?" How does it compare to the term "allocated size?" What's the relationship between the two?

ii. What is the `OurStack()` function? Why is each line in that function necessary?

iii. What is the `~OurStack()` function? Why is each line in that function necessary? Why isn't there any code in there involving the `logicalSize` or `allocatedSize` data members?

iv. In the `OurStack::grow()` function, one of the lines is `delete[] elems`, and in the next line we immediately write `elems = newArr;`. Why is it safe to do this? Doesn't deleting `elems` make it unusable?

v. The `OurStack::pop()` function doesn't seem to have any error-checking code in it. What happens if you try to pop off an empty stack?

vi. What is the significance of placing the `OurStack::grow` function in the private section of the class? What does that mean? Why didn't we mark it public?

Right now, our stack doubles its allocated length whenever it runs out of space and needs to grow. The problem with this setup is that if we push a huge number of elements into our stack and then pop them all off, we'll still have a ton of memory used because we never shrink the array when it's mostly unused.

vii. Explain why it would not be a good idea to cut the array size in half whenever fewer than half the elements are in use.

viii. Update the code for `OurStack` so that whenever the logical length drops below one quarter of the allocated length, the array is reallocated at half its former length. As an edge case, ensure that the allocated length is always at least equal to the initial allocated capacity. This setup maintains the amortized O(1) cost of each insertion and deletion and ensures that the memory usage is always O(n).