Section Solutions 5

Problem One: Pointed Points about Pointers

The output of the program is shown here:

```
0: 137, 0
1: 137, 10
2: 137, 20
3: 137, 30
4: 137, 40
0: 137, 0
1: 137, 10
2: 137, 20
3: 137, 30
4: 137, 40
0: 137, 0
1: 137, 10
2: 137, 20
3: 137, 30
4: 137, 40
```

Remember that when passing a pointer to a function, *the pointer is passed by value!* This means that you can change the contents of the array being pointed at, because those elements aren't copied when the function is called. On the other hand, if you change *which* array is pointed at, the change does not persist outside the function because you have only changed the copy of the pointer, not the original pointer itself.

Problem Two: Cleaning Up Your Messes

The first piece of code has two errors in it. First, the line

```
baratheon = targaryen;
```

causes a memory leak, because there is no longer a way to deallocate the array of three elements allocated in the first line. Second, since both `baratheon` and `targareon` point to the same array, the last two lines will cause an error.

The second piece of code is perfectly fine. Even though we execute

```
delete[] stark;
```

twice, the array referred to each time is different. Remember that you delete *arrays*, not *pointers*.

Finally, the last piece of code has a double-delete in it, because the pointers referred to in the last two lines point to the same array.

Problem Three: Creative Destruction

The ordering is as follows:

- A constructor is called when `elem` is declared in `main`.
- A constructor is then called to set `toPrint` equal to a copy of `elem`.
- A constructor is then called to initialize the `temp` variable in `printStack`.
- When `printStack` exits, a destructor is called to clean up the `temp` variable.
- Also when `printStack` exits, a destructor is called to clean up the `toPrint` variable.
- When `main` exits, a destructor is called to clean up the `elem` variable.

Problem Four: Random Bag Grab Bag

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

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

The `public` keyword indicates that the member functions listed underneath it are publicly accessible by anyone using the `RandomBag` class. This essentially means that they form the public interface for the class.

The `private` keyword indicates that the data members listed underneath it are private and only accessible by the class itself. This means that those data members are part of the private implementation of the class and aren't something that clients should be touching.

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

It's the ***scope resolution operator***. It's used to indicate what logical part of the program a given name belongs to. The case we'll primarily see it used is in the context of defining member functions in a .cpp file, where we need to indicate that the functions we're implementing are actually member functions of a class, not freestanding functions.

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

It indicates that those member functions aren't allowed to change the data members of the class. Only `const` member functions can be called on an object when in a function that accepts an object of that class by `const` reference.

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

The worst-case time complexity of an operation is $O(n)$, which happens when we remove the very first element from the `Vector`. Our best-case complexity is $O(1)$ if we remove from the end of the `Vector`. On average, the runtime is $O(n)$, since on average $n/2$ elements need to get shifted over.

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?

> The worst-case complexity would be $O(n^2)$, which would happen if we get very unlucky and always remove the very first element of the `Vector`, making the total work done roughly equal to $(n\text{-}1) + (n\text{-}2) + (n\text{-}3) + \ldots + 2 + 1 = O(n^2)$. The best-case complexity would be $O(n)$, which happens if we always remove the first element of the `Vector` ($n \times O(1) = O(n)$). The average-case time complexity is $O(n^2)$, since on average each removal does $O(n)$ work and we do it n times.

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.

There a couple of different ways to do this. One option is based on the insight that removing from the very end a `Vector` is an O(1)-time operation, so if we remove the last element of the `Vector` at each step we'll get an O(1) worst-case bound. The problem is that the last element is decidedly not a random element, since it always holds the last thing added. However, we can easily fix this by simply choosing a random element of the array and swapping it with the one at the end. This makes the element at the end randomly-chosen, which we still need to do, but makes deletions run in time O(1). Here's some code for this.

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

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

    swap(elems[index], elems[elems.size() - 1]);
    elems.remove(elems.size() - 1);
    return result;
}
```

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.

Part of the challenge here is that we need to find a way to determine what the next element to be removed is going to be, but we have to do so in way that gives consistent results from call to call.

There are many different ways we can do this. One option, which guarantees a uniformly-random selection of elements from the `RandomBag`, is to store an extra variable keeping track of the index of the next element to remove. Every time we add or remove an element, we'll update this value to hold a new random value. Here's one way we can do this. First, the changes in the header:

```
class RandomBag {
public:
    void add(int value);
    int removeRandom();
    int peek() const; // <-- Don't forget to make this const!

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

private:
    Vector<int> elems;
    int nextIndex;
};
```

Next, the changes to the `.cpp` file. We've moved a lot of the logic out of `RandomBag::removeRandom` into `RandomBag::peek` in order to unify the code paths and avoid duplicating our logic.

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

    /* Stage a new element for removal. */
    nextIndex = randomInteger(0, elems.size() - 1);
}

int RandomBag::peek() const {
    if (isEmpty()) {
        error("That which is not cannot be!");
    }
    return elems[nextIndex];
}

int RandomBag::removeRandom() {
    int result = peek();

    swap(elems[nextIndex], elems[elems.size() - 1]);
    elems.remove(elems.size() - 1);

    /* Stage a new element for removal. */
    nextIndex = randomInteger(0, elems.size() - 1);
    return result;
}
```

Problem Five: Stackity Stack

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?

The logical size of the stack is the actual number of elements stored in the stack, whereas the allocated size is the length of the elems array. The logical length can be any number between 0 (inclusive) and allocatedSize (exclusive) depending on how many pushes and pops have been made.

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

This is the constructor. It's responsible for setting all the variables in the OurStack class to a reasonable set of defaults. The lines here set up the actual memory where the elements will be stored, configure the initial logical length (0 elements stored), and the initial allocated length (which is the default initial size given by the constant.)

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?

This is the destructor. Its job is to deallocate any extra memory or resources that were allocated by the `OurStack` class and not otherwise automatically reclaimed. Here, we deallocate the memory we allocated earlier by using `delete[]`. We don't need to do anything to `logicalSize` or `allocatedSize` because there were no dynamic allocations performed to get space for them. Generally speaking, you only need to explicitly clean up resources that you explicitly allocated.

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?

The statement `delete[] elems;` means "destroy the memory *pointed at by* ***`elems`***" rather than "destroy the ***`elems`*** *variable*." As a result, after the first line executes, `elems` is still a perfectly safe variable to reassign.

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?

Notice that `OurStack::pop` calls `OurStack::peek`, which in turn has its own error-checking. As a result, trying to pop off an empty stack will trigger the custom error message from the `OurStack::peek` member function.

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?

> This is a private member function. This is a member function that's used as a part of the implementation of the class rather than the interface. Any clients of `OurStack` can't access this function, which is a Good Thing because we don't want rando's coming along and increasing our stack's capacity unnecessarily.

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.

> Imagine we have a stack with logical size $n - 1$ and allocated size n. If we perform two pushes, we'll double our stack's allocated size to $2n$, which takes time $O(n)$, and increase the logical size to $n+1$. If we now do two pops, we'll drop the logical size down to $n-1$, which is less than half the allocated size. If we now shrink the allocated size by half down to n, we'll have to do $O(n)$ work transferring elements over, ending with a stack with logical size $n - 1$ and allocated size n. Overall, we've done two pushes and two pops, we're right back where we've started, and we've done $O(n)$ work. That's a lot of work for very little payoff!

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.

There are many ways we can do this. We're going to do this by replacing the `OurStack::grow` function with a new `OurStack::setCapacity` function that lets us say what the new allocated size will be. Here's the updated class definition:

```
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 setCapacity(int capacity);

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

Here's the updated functions from the .cpp file:

```
void OurStack::push(int value) {
    if (logicalSize == allocatedSize) {
        setCapacity(allocatedSize * 2);
    }

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

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

    if (logicalSize < allocatedSize / 4) {
        setCapacity(allocatedSize / 2);
    }

    return result;
}

void OurStack::setCapacity(int capacity) {
    /* Never allocate fewer than kInitialSize cells in our array. */
    capacity = max(capacity, kInitialSize);

    int* newArr = new int[capacity];
    for (int i = 0; i < size(); i++) {
        newArr[i] = elems[i];
    }

    delete[] elems;
    elems = newArr;
    allocatedSize = capacity;
}
```