

Section Solutions 5

Problem One: Constructors and Destructors

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 Two: Writing Destructors

The destructor is called for an `IntStack` when that object is being destroyed. After the destructor finishes running, nothing in the program will ever be able to access that object again. Accordingly, there's no reason to change the variables, because no one will ever be able to read their values.

Problem Three: Pointer Traces!

The output of the program is

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

<pre>int main() { int* baratheon = new int[3]; int* targaryen = new int[5]; baratheon = targaryen; targaryen = baratheon; delete[] baratheon; delete[] targareon; }</pre>	<pre>int main() { int* stark = new int[6]; int* lannister = new int[3]; delete[] stark; stark = lannister; delete[] stark; }</pre>	<pre>int main() { int* tyrell = new int[137]; int* arryn = tyrell; delete[] tyrell; delete[] arryn; }</pre>
---	--	---

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.