CS 106B Section 6 (Week 7) Solutions

---

1. Writing Destructors

The destructor is called for a **`VectorInt`** 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.

---

2. New[] and delete[]

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

  baratheon = targaryen;
  targaryen = baratheon;

  delete[] baratheon;
  delete[] targaryen;
}
```

```
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;
}
```

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 **targaryen** 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.

CS 106B Section 6 (Week 7) Solutions

3. Pointer tracing

```
0: 0, 0
1: 1, 10
2: 2, 20
3: 3, 30
4: 4, 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.

---

4. Hashing (part 1)

hash1 is valid, but not good because everything will get hashed to the same bucket.
hash2 is not valid, because “A” and “a” are equal, but will have different hash values.
hash3 is valid and good.
hash4 is not valid, because equal strings might not give the same hash value.

---

5. Hashing (part 2)

Bucket:

```
0: baggage (30)
1: badcab (13)
2: feed (20) -> deadbeef (32)
3: cafe (15) -> cabbage (21)
4: null
5: null
```