CS 106B Section 6 (Week 7)

As you begin to dynamically allocate memory, you will see that memory management is very important. This handout will go over pointers and how to avoid memory leaks! We will also review hashing and how hashing functions work in practice.

Recommended problems: #3, #4

---

1. Writing Destructors.

In lecture, we wrote a **`VectorInt`** class that represents a vector of ints using an internal array. The private data members of the class were:

```
private:
  int* data;
  int size;
  int allocatedSize;
```

The destructor for the class is defined as follows:

```
VectorInt::~VectorInt() {
  delete[] data;
}
```

Why doesn't the destructor do anything to **size** or **allocatedSize**? Wouldn't it make sense to set them to 0?

---

2. 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;
}
```

CS 106B Section 6 (Week 7)

3. Pointer tracing.

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 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;
    }

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

    delete[] one;
    delete[] two;
    return 0;
}
```

4. Hashing (part 1).

Let's say we have a class `StRiNg` where two `StRiNgs` are considered equal if they are equal, ignoring upper and lower case. Other than that, they are the same as normal strings. Which of the following functions are legal hash functions for `StRiNgs`? Which functions are *good* hash functions?

```
int hash1(StRiNg& s) {
  return 0;
}
```

```
int hash3(StRiNg& s) {
  int product = 1;
  for (int i = 0; i < s.length(); i++) {
    product *= tolower(s[i]);
  }
  return product;
}
```

```
int hash2(StRiNg& s) {
  int sum = 0;
  for (int i = 0;
       i < s.length(); i++) {
    sum += s[i];
  }
  return sum;
}
```

```
int hash4(StRiNg& s) {
  return (int) &s;
}
```

5. Hashing (part 2).

If our hash table has 6 buckets, diagram the result of putting the following values into the hash table, using a hash function that adds up the values of each letter in the string (where 'a' is 1, 'b' is 2, etc.) and mods by the hash table length (6). If two strings collide, put them into a linked list.

`cabbage, baggage, deadbeef, cafe, badcab, feed`