More Advanced Classes

and lastly… “The Big Three”

Review

- Talked about constructor initialization
- Showed how to pass objects
- Recall: pass by reference is more efficient than pass by value
  - However, any changes will apply to the original object
  - Use **const** to tell the compiler:
    “do not allow changes to any member variables”

Review of Destructor

- Deletes the object
- Will be automatically created if one is not supplied
  - Will not handle dynamic memory

Example prototype:

```
~Class_Name(); //no return type, no parameters, can only have 1
```

- Called when the object goes out of scope
  - When the function ends
  - When the program ends
  - A block containing local variables ends
  - A delete operator is called

static – another keyword

- Creates a variable shared by all the instances of the same class

```
static int count;
```

- Allowed to be private
- Permits objects of the same class to communicate
- Important detail: must be initialized outside of class

```
int Point::count = 0;
```

  - The person writing the class does this
  - Static variables can’t be initialized twice

Another use of **static**

- Can have static functions
  - Are not attached to an object
  - There is no “this” pointer
  - Consequently… you can’t change any non-static member variables

```
static int getCount();

int Point::getCount() { //note: no static keyword
  count++;
  return count;
}
```

Function call: `Point::getCount();`

Shallow Copy vs. Deep Copy

- Shallow:
  - Also known as: **member-wise copy**
  - Copy the contents of member variables from one object to another
  - Default behavior when objects are copied or assigned
- Deep:
  - Copy what each member variable is pointing to so that you get a separate but identical copy
  - Must be programmer-specified

Copy Constructor

- This is what allows us to create a new copy of an object
- Constructor that has one parameter that is of the same type as the class
  - **Must accept reference** as parameter (normally const)
    - Why is this?

```
Point(const Point &); // prototype for a copy constructor
```

- Called automatically in three cases:
  - When a class object is being declared and initialized by another object of same type

```
Point CoolPoint = myPoint;
```

  - Whenever an argument of the class type is “plugged in” for a call by value parameter

```
passByValue(myPoint);
```

  - When a function returns a value of the class type

```
return myPoint;
```

Assignment Operator Overload

- This is invoked when we set an **existing instance** equal to some other instance of a class

```
Point& operator=(const Point&); // prototype for = overload
```

- Should return a reference to the updated instance
  - Allows us to chain assignments together: e.g. a = b = c
    - First set “b = c” and return a reference to b. Now set “a = b”
  - Need to make sure the assignment operator returns something of the same type as its left hand side

The Big Three

- If you implemented either a **Destructor**, a **Copy Constructor**, or a **Copy Assignment Operator**, you should ensure that all 3 are defined.
- If you needed one, you probably need all of them.
- This rule of thumb goes by several names:
  - The Big Three
  - The Rule of Three
  - The Law of The Big Three
- In C++11 there's an expanded version: The Big 5
  - We won't cover this yet