Homework #8 - Operator Overloading

Name:

1 Changing the Meaning of Operators

We talked last class about the general idea of a **class**, which is a collection of data and functions stored together in the computer's memory, inside a single variable. We talked about how - in general - we write "set" and "get" functions to access data inside our classes. Well, what do we do if we want to use built-in operations like "+", "-", "*", or "/" with our classes? What if we want to be able to print our classes out nicely using "<<"? This can be done with something called **operator overloading**. Basically, we can teach the computer a **new meaning** for operations like "+" or "<<", and this can make our classes even easier to work with. Here is an example of a Point class for storing 2D points (x, y):

```
class Point {
    // Data
    private:
      int x,y;

    // Methods (functions inside the class)
    public:
        // Empty constructor
        Point();

        // Constructor
        Point(int xp, int yp);

        // "+" operator overload
        Point operator+(const Point& otherPoint);

        friend ostream& operator<<(ostream& os, const Point& pt);

        void setPoint(int xp, int yp) {
            x = xp;
            y = yp;
        }

        int getX() {
            return x;
        }

        int getY() {
            return y;
        }
};
```

operatorOverloading.cpp

It looks a lot like the classes we used before. It still has "set" and "get" functions, but there's some other new stuff as well. Let's talk about that.

1.1 Constructors

You'll notice that there are two functions in the class that look like this:

```
// Empty constructor
Point();

// Constructor
Point(int xp, int yp);
```

These are called **constructors**. Their job is to set up everything inside the class each time we make a new copy of it (a new copy of a class is called an **object**). When we first make an **object** we sometimes do it like this:

```
Point p;
```

And sometimes we want to do it in a way that sets up all the data inside the object, like this:

```
Point p = Point(2,5);
```

This sets up a new Point with $x = 2$ and $y = 5$. This is why there are **two** constructors: the "empty" one handles cases where we don't need to set up the data inside our object, or when we aren't making a new object yet, and the non-empty one handles cases where we are making a new object and need to set it up.

1.2 Overloading Functions

So why are we allowed to have two different functions with the same name? We have two **constructors**, after all:

```
// Empty constructor
Point();

// Constructor
Point(int xp, int yp);
```

They are both called "Point", so how does the computer know which one we want to use? In C++, you are allowed to have more than one function with the same name, as long as the functions have different numbers of **arguments** (the things inside the parentheses "()"). Here, we have one constructor that takes two arguments, and one that takes none. This idea is called **overloading**. Basically, when the computer is making our program into machine code, it can tell which function is the right one to use by checking to see how many **arguments** and what **type** of arguments we have put inside the "()".

1.3 Prototypes vs Definitions

Well hold on, this just keeps getting stranger! These two constructors do not contain any code! There's no "{}". They simply end with ";" like this:

```
// Constructor
Point(int xp, int yp);
```

So how does this function work? It doesn't actually do anything! In C++, you can actually split functions into two parts. One part is called a **prototype** and one part is called a **definition**. The **prototype** just tells us what kind of **arguments** go into the function. Later on in our code, we have to also write a **definition** which says says how the function will actually work. The strange thing about **constructors** is that their **definitions** go outside the class. So later on in our code, we have:

```
// Empty constructor
Point::Point() {

}

// Tell the computer how to make a new point
Point::Point(int xp, int yp) {
    x = xp;
    y = yp;
}
```

operatorOverloading.cpp

These actually tell the computer what the two **constructors** should do. The "Point::" part in front just tells the computer that these two functions belong to the class "Point".

1.4 Overloading Operators

We need the idea of a **prototype** for operator overloading. If you look back at the first page, you'll see that there's a **prototype** for the "+" operator inside our Point class...but there's no code saying how it works. That's farther down in our program code and looks like this:

```
// Tell the computer what "+" means for the Point class
Point Point::operator+(const Point& otherPoint) {
    Point newPoint(x+otherPoint.x, y+otherPoint.y);
    return newPoint;
}
```

operatorOverloading.cpp

The "+" operator has to be **defined** outside of the Point class so that we can use it without also saying the name of the class (like this code to make a new point):

```
Point x = Point.Point()
```

Putting the **definition** outside the class keeps us from having to say "Point.Point()" every time we want to use the constructor, or "Point.+(otherPoint)" every time we want to add points together.

2 Your Turn

2.1 Overloading Math

Change your Array class so that it overloads "+", "-", "*", and "/" for arrays (so that you can multiply all the numbers in two arrays together just by doing "A * B", for instance). You only need to make this work for arrays that are the same length.

2.2 Overloading output operators

Look at the example code for the Point class and notice that "<<" is overloaded too. Try to overload this in your Array class as well, so that you can do things like this:

```
1 cout << A << endl;
```