Operator Overload

Nov 10, Ch 11.1 - 11.3
Announcements

Last lab (8) will be taken out of 5 points (although there were 6 problems)
Highlights

- operator overload

```cpp
class Point{
public:
    friend bool equals(Point first, Point second);
};
```
Q: When do you use classes?
A: When you have multiple variables that are closely related

For example, a longitude and latitude could simply be two separate doubles

But being together in a class, you to can pass them around together and manipulate easier
A class constructor is used to initialize all the variables inside the class (member variables).

Constructors and classes are linked:
- It is impossible to create a class instance without using a constructor
- Every constructor makes one class instance

Sometimes C++ assumes a default constructor exists (and will try to use it)
Operator overloading

Consider this simple class:

```cpp
class Point{
private:
    int x;
    int y;
public:
    Point();
    Point(int startX, int startY);
    void showPoint();
};
```

(See: pointClass.cpp)
Operator overloading

If we have an \((x,y)\) point, it would seem natural that we could add two of them.

We can overload the `+` operator to allow easy addition of points.

```cpp
Point operator+(Point other)
{
    Point result;
    result.x = x + other.x;
    result.y = y + other.y;
    return result;
}
```

(See: `pointOverload.cpp`)
Operator overloading

When overload operators in this fashion, the computer will convert a statement such as:

```plaintext
Point c = a+b;
```

... into ...

```plaintext
Point c = a.operator+(b);
```

... where the left side of the operator is the “calling” class and the right side is a argument
Operator overloading

You cannot change the number of parts to an operator ('+' only gets 2, '!' only gets 1)

Cannot create “new” operators (can only overload existing ones)

Cannot change order of precedence ('*' is always before '+')

Operator '=' is special... save for later
friend functions

You can give a non-class function access to private variables by making it a friend

Friends don't mind sharing!
friend functions

Instead of declaring a friend function at the top, do it inside the class:

```cpp
class Point{
public:
    friend bool equals(Point first, Point second);
}
```

The function description/implementation is identical to as if it was a non-friend:

```cpp
bool equals(Point first, Point second) {
}
```

(See: pointFriends.cpp)
friend functions

When would you want to use friend functions?

1. Typically when we want to involve two separate classes

2. When we care about the order of things...

For example, how would you overload '<<'?
(cout is of type “ostream”)
(See: pointOverloadCout.cpp)
What is the difference between these two?

```c
int sum(int x, int y);
int sum(const int &x, const int &y);
```
What is the difference between these two?

```cpp
int sum(int x, int y);
int sum(const int &x, const int &y);
```

First one copies the values into `x` and `y`, thus these values exist in multiple places.

The second creates a link but does not let you modify the original.
Classes can be rather big, so in this case using const and ' &' can save memory.

So a better way to write:

```cpp
bool equals(Point first, Point second)
```

... would be: (function definition the same)

```cpp
bool equals(const Point & first, const Point & second)
```
Average class

Let's do a larger example of a class (like a lab)

Suppose we want to make a class to find the average of an array of doubles, and want:
- A default constructor
- A constructor that takes an array and size
- Overload '<<' for cout and average
- Overload '+=' to average over another double
- Overload '+' to add two averages together
(See: average.cpp)
Simple card game

Let's make a small card game based on hearthstone (simplified)
Simple card game

Rules:
- Cards **attack** and **defense** values
- Defender draws one card
- All cards always attack
- Defender selects which card to defend with
- Attacker dies if attack < defense (vice versa)

(See: simpleStone.cpp)