Today

- Quiz 1
- C++ Operator Overloading (cont)

Assignments

- HW-1 out
**Relational Comparison Operators**

Adding `operator<` to a Rectangle class ...

```cpp
class Rectangle
{
    public:
        ... 
        bool operator<(const Rectangle& rhs) const;
    ... 
}
```

- in a call `r1 < r2` ...
- the **lhs** is `r1` (which is the current object) and the **rhs** is `r2`

And then in the implementation file ...

```cpp
bool Rectangle::operator<(const Rectangle& rhs) const
{
    return area() < rhs.area();
}
```

**Exercise:** Implement `>` and `==` without using `area()` ... i.e., use `<` and `*this`

```cpp
bool Rectangle::operator>(const Rectangle& rhs) const
{
    return rhs < *this;
}
```

```cpp
bool Rectangle::operator==(const Rectangle& rhs) const
{
    return !(*this < rhs) && !(rhs < *this);
}
```

You can implement **all** the relational comparators with only `<` !!!
**Arithmetic Operators**

Adding `operator+` to `Rectangle`

What should the behavior of `operator+` be? ... eg: `r3 = r1 + r2`;

- ... takes a `rhs` rectangle (`r2`)
- ... adds lengths and widths of `*this` (`r1`) and `rhs` (`r2`)
- ... creates a new rectangle object (to hold the result of the addition)
- ... returns the new rectangle (copied into `r3`)

The signature:

```cpp
class Rectangle
{
    public:
        ...
        Rectangle operator+(const Rectangle& rhs) const;
    ...
};
```

Q: How can we implement `operator+`?

```cpp
Rectangle Rectangle::operator+(const Rectangle& rhs) const
{
    Rectangle result;
    result.length = length + rhs.length;
    result.width = width + rhs.width;
    return result;
}
```
Unary Operators

Adding (unary minus) operator- to “rotate” a Rectangle

Similar to addition, doesn’t modify the object, gives a new object … e.g.:

```cpp
int x = 4;
int y = -x;
cout << x << endl; // prints 4
cout << y << endl; // prints -4
```

Defining the overloaded operator

```cpp
class Rectangle
{
  public:
    ...
    Rectangle operator -( ) const;
    ...
};
```

And the implementation …

```cpp
Rectangle Rectangle:: operator -( ) const
{
  Rectangle result;
  result.width = length;
  result.length = width;
  return result;
}
```
**Increment (++) and Decrement (--) operators**

Recall the difference between prefix and postfix ++

```cpp
int x = 1;
int y = ++x;
int z = x++;
cout << "x = " << x << endl; // prints 3
cout << "y = " << y << endl; // prints 2
cout << "z = " << z << endl; // prints 2
```

- increment modifies value of `x` ... not true with unary minus and addition
- prefix version increments `x`, returns incremented value
- postfix version returns `x`, then increments `x`

postfix requires more work (more operations) ...

```cpp
class Rectangle
{
    public:
        ...
        Rectangle& operator++(); // prefix version
        Rectangle operator++(int); // postfix version
        ...
};
Rectangle& Rectangle::operator++()
{
    ++length;
    ++width;
    return *this;
}
Rectangle Rectangle::operator++(int)
{
    Rectangle tmp = *this;       // create a temporary copy
    ++length;
    ++width;
    return tmp;
}
```
**Overloaded operators as non-member function (friends)**

A **friend** function of a class is:

- a non-member function
- that can access protected & private class members

Example:

```cpp
class Rectangle {
    public:
        ...
        friend Rectangle operator+(const Rectangle& lhs, const Rectangle& rhs);
        ...
};

// the function implementation ...

Rectangle operator+(const Rectangle& lhs, const Rectangle& rhs) {
    Rectangle result;
    result.length = lhs.length + rhs.length;
    result.width = lhs.width + rhs.width;
    return result;
}
```

**Overloaded operators can always be implemented outside the class**

- provides greater control of lhs versus rhs operand types
- e.g., a `<` operator to compare rectangles and circles
- don’t need to be friend functions unless access protected/private members
Stream insertion and extraction operators

Stream insertion (<<) and extraction (>>) must be non-member functions!

- e.g., for: `cout << r1;` or `cin >> r1;`

Example

```cpp
class Rectangle
{
public:
  ...

  friend std::ostream& operator<<(std::ostream& out, const Rectangle& rhs);

  friend std::istream& operator>>(std::istream& in, Rectangle& rhs);
};
```

// implementation ...

```cpp
std::ostream& operator<<(std::ostream& out, const Rectangle& rhs)
{
  out << rhs.length << " " << rhs.width;
  return out;
}

std::istream& operator>>(std::istream& in, Rectangle& rhs)
{
  in >> rhs.length;
  in >> rhs.width;
  return in;
}
```

Q: Why do we return the streams?

Q: Explain the signatures (const, references, return types)