Operator Overloading

Section 11.6
Skipping 11.3, 11.4, 11.5 for now
Purpose of Operator Overloading

Operator overloading “extends” the operators currently defined in the language.

Example: How can we add two Rational numbers? Presently the following is not possible because the == operator is not defined for Rational numbers.

Rational cR1 (2,1), cR2 (2,3);
if (cR1 == cR2)
{
    cout << “Fractions are equal”;  
}
Comparing two Rational numbers

class Rational
{
    public:
        Rational(int = 0, int = 1);
        Rational addition(const Rational &);
        Rational subtraction(const Rational &);
        Rational multiplication(const Rational &);
        Rational division(const Rational &);
        void printRational() const;
    private:
        int numerator;
        int denominator;
        int greatestCommonDivisor (int numOne, int numTwo);
        void reduce();
};

Question: How would we add isEqual to Rational?
Rational “is equal” Solution #1

Add the following method to Rational.h

```cpp
bool isEqual (const Rational &) const;
```

Add the following code to Rational.cpp

```cpp
bool Rational::isEqual (const Rational &cRational) const
{
   return (numerator == cRational.numerator &&
            denominator == cRational.denominator);
}
```

How do we use isEqual?

```cpp
Rational cR1 (2, 3), cR2 (2, 3);
if (cR1.isEqual (cR2))
{
   ....
}
```
Rational “is equal” Solution #2

Add the following code to Rational.h

```cpp
int operator==(const Rational &rop2) const
{
    return (numerator == rop2.numerator) &&
           (denominator == rop2.denominator);
}
```

The above code is “overloading” the == operator to be able to compare two Rational objects; thus, the following code is now legal:

```cpp
if (cR1 == cR2)
{
    ....
}
```
Add the following code to Rational.h

```cpp
int operator== (const Rational & ) const;
```

Add the following code to Rational.cpp

```cpp
int Rational :: operator== (const Rational& rop2) const
{
    return (numerator == rop2.numerator) &&
    (denominator == rop2.denominator);
}
```

What you notice is that any binary operator whether it be arithmetic, relational, or logical uses the first operand as a default and the second operand corresponds to the one formal parameter.
Common Mistake

You might think the statement:

```cpp
int operator== (const Rational& rop2) const;
```

should look like:

```cpp
int operator== (const Rational& rop1,
               const Rational& rop2) const;
```

but that is not the case. The reason is that the operator function is defined as a member of the class and as such, one of the arguments for the operator is implicitly the first object of the operation. Make a note of this because the compilation error might be hard to fix.