Operator Overloading

Writing a Rational Number Class

- Operator == and !=
- Operator +
- Operator ++
- Operator double

Overloading Operators

Most operators can be overloaded in classes where at least one operand is a class type

Some operators that cannot be overloaded:

.	member selector
::	scope resolution
.*	pointer to member
?:	conditional operator

Rules for Overloading Operators

You cannot change:

precedence	(* before +)
number of operands	(++ has 1 operand)
associativity	(2+3+4 from left)

Some operators must be class member functions:
( ) [ ] -> =

Other operators can be member or non-member functions

Rational Number Class

A rational number is a number in form
a / b where a, b are integers and b != 0

Ex. 1 / 3

A rational number is stored in lowest terms:
2 / 4 is stored as 1 / 2

A negative rational number stores the negative value in the numerator

Ex. -1 / 3

```
public:
    Rat();                          // POST: 0 / 1 is constructed
    Rat(int n, int d);              // PRE: d != 0
                                    // POST: n / d is constructed
    // accessors and modifiers
    int getNum() const;
    int getDen() const;
    void setNum(int n);
    void setDen(int d);             // PRE: d != 0

    // relational operators
    bool operator==(const Rat& r) const;    // POST: return true if object == other
    bool operator!=(const Rat& r) const;    // POST: return true if object != other

    // math operators
    Rat operator+(const Rat& r) const;      // POST: return rational that is object + r
    Rat operator++();                       // POST: prefix: add 1, return updated object
    Rat operator++(int dummy);              // POST: postfix: add 1, return original object

    // type conversion
    operator double();                      // POST: return double quotient
};

// IO operators
ostream& operator<<(ostream& stream, const Rat& r); // POST: display n / d on output stream
istream& operator>>(istream& stream, Rat& r);       // PRE: denominator != 0
                                                    // POST: read n d from input stream
```

Rational Number Class

```
#pragma once
#include <iostream>
using namespace std;

class Rat
{   private:
        int num;                    // numerator
        int den;                    // denominator
        int gcd(int n, int d);      // POST: return greatest common divisor n, d
```

Function gcd is a private class member to compute the greatest common divisor to reduce a fraction to lowest terms

2 / 4	reduces to	1 / 2	gcd = 2
24 / 36	reduces to	2 / 3	gcd = 12

Greatest Common Divisor

```
#include "Rat.h"
#include <cmath>
using namespace std;

int Rat::gcd(int n, int d)                  // POST: return greatest common divisor n, d
{   n = abs(n);                             //  get absolute value of n
    d = abs(d);                             //  get absolute value of d
    int result = 1;                         //  current gcd
    for (int k = 1; k <= n && k <= d; k++)  // continue while k <= smaller value
        if (n % k == 0 && d % k == 0)       // does k evenly divides both values?
            result = k;
    return result;
}
```

Constructors

```
Rat::Rat()                          // POST: 0 / 1 is constructed
{   num = 0;
    den = 1;
}

Rat::Rat(int n, int d)              // PRE: d != 0
{   int factor = gcd(n, d);         // POST: n / d is constructed
    if (d < 0) n = -n;              // move negative sign to numerator
    num = n / factor;               // reduce num and den
    den = abs(d) / factor;
}
```

Rat r (-5, -15);
n: -5
d: -15
factor: 5
d < 0 so n: 5
num: 5 / 5 = 1
den: abs(-15) / 5 = 3
Rational is 1 / 3

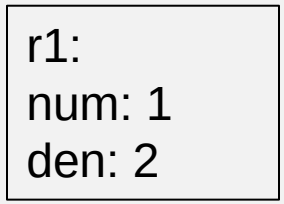
Rat r (4, -8);
n: 4
d: -8
factor: 4
d < 0 so n: -4
num: -4 / 4 = -1
den: abs(-8) / 4 = 2
Rational is -1 / 2

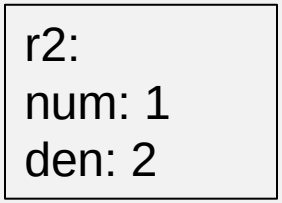
Rat does not use Dynamic Memory

The default empty destructor will be used

The default copy constructor will use a shallow memory copy which is sufficient

```
Rat r1 (1, 2);
Rat r2 (r1);
```





Accessors and Modifiers

```
int Rat::getNum() const          // POST: return numerator
{   return num;
}

int Rat::getDen() const          // POST: return denominator
{   return den;
}

void Rat::setNum(int n)          // POST: numerator is set
{   int factor = gcd(n, den);
    num = n / factor;
    den = den / factor;
}

void Rat::setDen(int d)          // PRE: d != 0
{   int factor = gcd(num, d);    // POST: denominator is set
    if (d < 0) num = -num;
    num = num / factor;
    den = abs(d) / factor;
}
```

Relational Operators

```
bool Rat::operator==(const Rat& r) const     // POST: return true if object == other
{    int n1 = num * r.den;                   // r.getDen() is also correct
     int n2 = r.num * den;
     return n1 == n2;
}

bool Rat::operator!=(const Rat& r) const     // POST: return true if object != other
{    return  !(*this == r);
}
```

```
Rat r1 (1, 2);
Rat r2 (3, 4);
if (r1 == r2) cout << "same";
```

n1: num * r.den 1 * 4 = 4
n2: r.num * den 3 * 2 = 6

The left operand r1 makes the function call and r2 is passed as the argument

Relational Operators

```
bool Rat::operator==(const Rat& r) const     // POST: return true if object == other
{    int n1 = num * r.den;                   // r.getDen() is also correct
     int n2 = r.num * den;
     return n1 == n2;
}

bool Rat::operator!=(const Rat& r) const     // POST: return true if object != other
{    return  !(*this == r);
}
```

```
Rat r1 (1, 2);
Rat r2 (3, 4);
if (r1 != r2) cout << "same";
```

r1 makes the call
*this is object r1
Call operator== passing r2 to parameter r

The left operand r1 makes the function call and r2 is passed as the argument

Mathematical Operators

```
Rat Rat::operator+(const Rat& r) const       // POST: return rational object + r
{    int n = num * r.den + den * r.num;
     int d = den * r.den;
     return Rat(n, d);
}
```

Rat r1 (1, 2);
Rat r2 (3, 4);
Rat r3;
r3 = r1 + r2;

n: 1 * 4 + 2 * 3 = 10
d: 2 * 4 = 8
Call parameterized constructor Rat (10, 8)
In this new object, gcd is called to reduce fraction
Return this object set to 5 / 4

Increment Operator

The prefix version adds 1 to the operand and returns the updated value

```
int k = 5;
int m = ++k;                    // m  stores 6
```

The postfix version adds 1 to the operand but returns the original value

```
int k = 5;
int m = k++;                    // m  stores 5
```

Increment Operator

```
Rat Rat::operator++()              // POST: prefix version
{    num = num + den;              // add 1 to object, return updated value
     return *this;
}

Rat Rat::operator++(int dummy)     // POST: postfix version
{    Rat temp(num, den);           // add 1 to object, return original value
     num = num + den;
     return temp;
}
```

Add a parameter though it is not used

```
Rat r1 (1, 2);
Rat r2 (3, 4);
Rat r3 = ++r1;
Rat r4 = r2++;
```

1 / 2 + 1 / 1 = 1 / 2 + 2 / 2 = 3 / 2
3 / 4 + 1 / 1 = 3 / 4 + 4 / 4 = 7 / 4

Type Conversion Operator

Overload the double operator to convert a Rat object to a double value

```
Rat::operator double()          // POST: return double quotient
{   return  (double) num / den;
}
```

```
Rat r1 (1, 2);
double d = (double) r1;
```