Comp 5421 BB | Assignment 3 | Due by 9 July 2016

Objectives

1. To implement an abstract data type (ADT)[1]
2. To practice using the operator overloading facility of the C++ language
3. To learn about function objects and how to define them

Fraction ADT

Objects: Normalized fractions

Fractions of the form $\frac{a}{b}$ where both a and b are integers, with $b \neq 0$. The integers a and b are called the *numerator* and the *denominator* of the fraction, respectively. Normalized fractions have positive denominators; that is, $b > 0$.

Arithmetic operations

Addition: $\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}$

Subtraction: $\frac{a}{b} - \frac{c}{d} = \frac{ad-cb}{bd}$

Multiplication: $\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$

Devision: $\frac{a}{b} / \frac{c}{d} = \frac{ad}{bc}$

Negation: $-\frac{a}{b} = \frac{-a}{b}$

Inversion: $\mathit{inverse}\left(\frac{a}{b}\right) = \frac{b}{a}, \quad a \neq 0$

Normalization: $\mathit{normalize}\left(\frac{a}{b}\right) = \frac{-a}{-b}$ if $b < 0$

Reduction: $\mathit{reduce}\left(\frac{a}{b}\right) = \frac{a'}{b'}$ where $a = ga'$, $b = gb'$, $g = gcd(a, b)$

Relational operations

$\frac{a}{b} = \frac{c}{d}$ if and only if $ad = bc$

$\frac{a}{b} \neq \frac{c}{d}$ if and only if $!\left(\frac{a}{b} = \frac{c}{d}\right)$

$\frac{a}{b} < \frac{c}{d}$ if and only if $ad < bc$

$\frac{a}{b} > \frac{c}{d}$ if and only if $\frac{c}{d} < \frac{a}{b}$

$\frac{a}{b} \leq \frac{c}{d}$ if and only if $!\left(\frac{a}{b} > \frac{c}{d}\right)$

$\frac{a}{b} \geq \frac{c}{d}$ if and only if $!\left(\frac{a}{b} < \frac{c}{d}\right)$

The fact that the denominators of fractions are positive leads to simple definitions of the relational operations above.

[1] ADT = Objects + Operations − Implementation details

A fraction is said to be in *normalized* form if its denominator is positive. For example, the fraction $\frac{3}{-5}$ is not normalized but can be normalized to $\frac{-3}{5}$ where the numerator absorbs and represents the sign of the fraction.

A fraction is said to be *reduced* to lowest terms if its numerator and denominator have no common factors other than ± 1. Thus any fraction can be reduced to lowest terms by dividing both its numerator and denominator by their greatest common divisor (gcd). For example, since $5 = gcd(15, 20)$, the fraction $\frac{15}{20}$ reduces to the fraction $\frac{3}{4}$.

Your Task

Implement a **Fraction** class so that **Fraction** objects can be used as operands in integer arithmetic expressions.

Example 1: mixing integers with Fraction objects

```
1  Fraction_2016 f1;   // = (0/1)   = 0
2  Fraction_2016 f2(5);   // (5/1) = 5
3  Fraction_2016 f3(55, -77);   // = (-5/7), both normalized and reduced
4
5  // mixing ints and Fractions
6  Fraction_2016 expr = f1 + 1 - f2 * 2 + (-f3) / 1 + 10;
7  cout << "expr: " << expr << endl;
8  assert(expr == Fraction_2016(12, 7));
9  cout << "f1: " << f1 << endl;
10 cout << "f2: " << f2 << endl;
11 cout << "f3: " << f3 << endl;
12 cout << "victory!" << endl;
```

Output

```
13 expr: 12/7 = 1 + 5/7
14 f1: 0
15 f2: 5
16 f3: -5/7
17 victory!
```

Notice that class **Fraction** provides three ways to construct **Fraction** objects, as shown on lines 1-3. Class **Fraction** accomplishes the three constructions through a single constructor with two optional parameters of type **long**, using **0** and **1** as the default parameter values for the numerator and denominator, respectively.

Lines 3 and 16 show that that the fraction under construction is both normalized and reduced. The supplied numerator and denominator values must be adjusted into a normalized fraction so that it may later be used in a relational expression—recall that the definitions of relational operators presented on page 1 applies only to normalized fractions.

The reduction part of a newly constructed fraction is an attempt to minimize unnecessary arithmetic overflow as the values of the numerator and denominator can grow in size very rapidly:

An example of growth rate of numerator and denominator

without reduction $$\frac{1}{111}+\frac{1}{222}+\frac{1}{333}=\frac{333}{24642}+\frac{1}{333}=\frac{135531}{8205786}$$

with reduction $$\frac{1}{111}+\frac{1}{222}+\frac{1}{333}=\frac{333}{24642}+\frac{1}{333}=\frac{1}{74}+\frac{1}{333}=\frac{407}{24642}=\frac{11}{666}$$

Note that $\frac{135531}{8205786}=\frac{12321}{12321}\times\frac{11}{666}$

Thus, **Fraction** applies reduction whenever possible during an operation.

The expression on line 6 necessitates operator overloads that can perform operations of the forms $-f$, $f * i$, f/i, $f + i$ and $f_1 - f_2$, where f, f_1 and f_2 denote **Fraction** objects and i denotes an **int**.

Line 7 requires an operator overload that can write a **Fraction** object to **cout**, and line 13 shows how it should look like when the numerator is $\geq$ the denominator. Lines 14-15 show how a fraction like **f2** whose denominator is **1** should be written to **cout**: without the "/1" part.

Relative to your previous assignments, this assignment is easy and involves no dynamic storage management. It involves implementing about 40 operator overloads, most of which are one-liner functions. The **Fraction** class itself is rather simple, containing a pair of **long** data members for storing the numerator and denominator, a pair of accessors (getters) and mutators (setters) for each data member, and a single constructor with two optional parameters.

To be clear about which 40 operator overloads to implement, Tables 1 and 2 on pages 5 and 6, respectively, list the prototypes for the required member and non-member operator overloads. As a result, you save time and effort with figuring out the return and parameter types and with deciding whether to implement an operator overload as a member of the class or as a non-member function.

Although it is possible to turn objects of a class into function objects in many ways by overloading the function call operator **operator()** as many times as desired, **Fraction** objects are turned into function objects in only one way, as prototyped on Table 1 at line 16.

For example, for a **Fraction** object **f** that stores the fraction $\frac{-5}{7}$ the call **f()** should return the string "**(-5/7)**".

Table 1, line 17, prototypes the dereference operator **operator***, enabling **Fraction** objects to behave like pointers, although **Fraction** objects have nothing to do with pointers! For example, for object **f** from the above paragraph, the expression ***f** really is **f** itself, behaving like (but only pretending to be) a pointer!

Please bear in mind that the dereference and function call operators and indeed most operators do not necessarily have to be overloaded for a class unless they provide meaningful service. For example, does it make sense to overload the subscript **operator[]** for **Fraction** objects? Regardless of an answer, one can't afford to miss the opportunity here. In fact, you are going to overload **operator[]** in a way that can be useful but is not too common: rather than implementing the following common overloads

Common Prototypes for Operator[] Overloads

```
// assume that an even k signals a request for the numerator and an odd k for the denominator
long& operator[](int k);
const long& operator[](int k)const;
```

you are going to implement the subscript overloads prototyped as follows:

Less Common Prototypes for Operator[] Overloads

```
// [s] returns the numerator if s="top"; otherwise, returns denominator
long& operator[](const string &s);
const long& operator[](const string &s)const;
```

The idea here is to remind you the although **operator[]** takes a single argument, the argument itself is not limited to type **int** only and can in fact be of any type, such as **string**, for example. To see these operator overloads in action, see lines 250-260 on page 12.

Finally, this assignment provides a rather detailed test driver code which, for reference, is printed on pages 7-13.

Table 1: Member Operator Overload Prototypes

```
Fraction& operator=  (const int& val);  // f =  i
Fraction& operator+= (const int& val);  // f += i
Fraction& operator-= (const int& val);  // f -= i
Fraction& operator*= (const int& val);  // f *= i
Fraction& operator/= (const int& val);  // f /= i

Fraction& operator= (const Fraction& rhs);    // f  = f
Fraction& operator+= (const Fraction& rhs);   // f += f
Fraction& operator-= (const Fraction& rhs);   // f -= f
Fraction& operator*= (const Fraction& rhs);   // f *= f
Fraction& operator/= (const Fraction& rhs);   // f /= f

Fraction & Fraction::operator++();         // ++f
Fraction & Fraction::operator--();         // --f

string operator()() const; // returns the string version of this fraction
Fraction& operator*(); // make Fraction objects behave like pointers: returns *this

// [s] returns the numerator if s="top"; otherwise, returns denominator
long& operator[](const string &s);
const long& operator[](const string &s)const;

// The following operators are usually overloaded as members
Fraction operator+(const Fraction& rhs) const;   // +f
Fraction operator-(const Fraction& rhs)const ;   // -f

Fraction Fraction::operator++(int);        // f++
Fraction Fraction::operator--(int);        // f--
```

Table 2: Non-Member Operator Overload Prototypes

```cpp
Fraction operator+(const Fraction& lhs, const Fraction& rhs); // f + f
Fraction operator+(const Fraction& lhs, const int& val); // f + i
Fraction operator+(const int& val, const Fraction& rhs); // i + f

Fraction operator-(const Fraction& lhs, const Fraction& rhs); // f - f
Fraction operator-(const Fraction& lhs, const int& val); // f - i
Fraction operator-(const int& val, const Fraction& rhs); // i - f

Fraction operator*(const Fraction& lhs, const Fraction& rhs); // f * f
Fraction operator*(const Fraction& lhs, const int& val); // f * i
Fraction operator*(const int& val, const Fraction& rhs); // i * f

Fraction operator/(const Fraction& lhs, const Fraction& rhs); // f / f
Fraction operator/(const Fraction& lhs, const int& val); // f / i
Fraction operator/(const int& val, const Fraction& rhs); // i / f

bool operator==(const Fraction& lhs, const Fraction& rhs); // f == f
bool operator==(const Fraction& lhs, const int& val); // f == i
bool operator==(const int& val, const Fraction& rhs); // i == f
bool operator!=(const Fraction& lhs, const Fraction& rhs); // f != f
bool operator!=(const Fraction& lhs, const int& val); // f != i
bool operator!=(const int& val, const Fraction& rhs); // i != f
// ditto for operators <, <=, >, >=

istream& operator>> (istream& istr, Fraction& rhs);
ostream& operator<< (ostream& ostr, const Fraction& rhs);
```

Test Drive

Test Driver Code: page 1/7

```
#include <iostream>
#include <iomanip>
#include <string>
#include <cassert>
using namespace std;
#include "Fraction.h"

using namespace std;

int main()
{
cout << "Test Fraction and Fractional Computations\n";
cout << "-----------------------------------------\n\n";

cout << "testing default ctor with Fraction f0;\n";
Fraction f0;          // test default ctor

cout << "testing fraction == integer with f0 == 0\n";
int zero = 0;
assert(f0 == zero);   // test fraction == int
assert(f0 == 0);      // test fraction == int literal
cout << "f0: " << f0 << '\n';
cout << "ok\n\n";

cout << "testing 1-arg ctor with Fraction f1(5);\n";
Fraction f1(5);       // test 1 arg ctor
assert(5 == f1); // test integer = fraction
cout << "f1: " << f1 << "\n\n";
cout << "ok\n\n";

cout << "testing copy ctor with Fraction f2 = f1;"
   << '\n';
Fraction f2 = f1;      // test ctor
assert(f2 == f1);      // test fraction == fraction
cout << "f2: " << f2 << "\n\n";
cout << "ok\n\n";

cout << "testing fraction == integer with f1 == 5;\n";
assert(f1 == 5);    // test fraction == integer
cout << "ok\n\n";
```

Test Driver Code: page 2/7

```
cout << "testing fraction == fraction with f1 == f2;\n";
assert(f1 == f2);    // test fraction == fraction
cout << "ok\n\n";

cout << "testing fraction != fraction with !(f1 != f2);\n";
assert(!(f1 != f2));    // test fraction != fraction
cout << "ok\n\n";

cout << "testing 2 args ctor with Fraction half = "
      "Fraction(1, 2);\n";
Fraction half = Fraction(1, 2);      // 2 args ctor
cout << "half: " << half << "\n\n";

cout << "testing operator+ with f2 = f1 + half;\n";
f2 = f1 + half;      // test operator+
cout << "f2: " << f2 << "\n\n";

cout << "testing operator< with f1 < f2;\n";
assert(f1 < f2);             // operator <
cout << "ok\n\n";

cout << "testing operator<= with f1 <= f2;\n";
assert(f1 <= f2);             // operator <=
cout << "ok\n\n";

cout << "testing operator> with f2 > f1;\n";
assert(f2 > f1);             // operator >
cout << "ok\n\n";

cout << "testing operator>= with f2 >= f1;\n";
assert(f2 >= f1);             // operator >=
cout << "ok\n\n";

cout << "testing operator!= with  f2 != f1;\n";
assert(f2 != f1);             // operator !=
cout << "ok\n\n";

cout << "testing operator==, operator- with "
"f1 == f2 - half;\n";
assert(f1 == f2 - half);        // operator -
cout << "ok\n\n";
```

Test Driver Code: page 3/7

```

cout << "testing 2 args ctor with Fraction oneThird"
       " (1, 3);\n";
Fraction oneThird(1, 3);
cout << "oneThird: " << oneThird << "\n\n";

cout << "testing assignment=, binary  +, -, and unary - "
       "with  f2 = f1 + oneThird - ( - oneThird );\n";
f2 = f1 + oneThird - (-oneThird);
cout << "f2: " << f2 << '\n';
assert(f2 == Fraction(17, 3));
cout << "ok\n\n";

cout << "testing fractional expression with "
       "f2 = f1 - oneThird + ( - oneThird );\n";
f2 = f1 - oneThird + (-oneThird); // unary +
cout << "f2 *: " << f2 << '\n';
assert(f2 == Fraction(13, 3));
cout << "ok\n\n";


cout << "testing post++ with f2 = f1++;\n";
f2 = f1++;
cout << "f1 : " << f1 << '\n';
cout << "f2 : " << f2 << '\n';
assert(f1 == Fraction(6));
assert(f2 == Fraction(5));
cout << "ok\n\n";

cout << "testing pre++ with f2 = ++f1;\n";
f2 = ++f1;
cout << "f1 : " << f1 << '\n';
cout << "f2 : " << f2 << '\n';
assert(f1 == Fraction(7));
assert(f2 == Fraction(7));
cout << "ok\n\n";

cout << "testing post-- with f2 = f1--;\n";
f2 = f1--;
cout << "f1 : " << f1 << '\n';
cout << "f2 : " << f2 << '\n';
assert(f1 == Fraction(6));
assert(f2 == Fraction(7));
cout << "ok\n\n";
```

Test Driver Code: page 4/7

```
cout << "testing pre-- with f2 = --f1;\n";
f2 = --f1;
cout << "f1 : " << f1 << '\n';
cout << "f2 : " << f2 << '\n';
assert(f1 == Fraction(5));
assert(f2 == Fraction(5));
cout << "ok\n\n";

f1 = Fraction(6) / 10;
cout << "testing normalization with f1 == \"6/10\";\n";
assert(f1 == Fraction(3, 5));
cout << "f1 : " << f1 << '\n';
cout << "ok\n\n";

cout << "computing sum = 1/3 + 1/3 + 1/3 + 1/3 + 1/3 +"
                 " 1/3 + 1/3 + 1/3 + 1/3 + 1/3\n";
Fraction sum;     // sum = 0
for (int k = 0; k < 10; ++k)
{
  sum = sum + oneThird;
  cout << "sum: " << sum << '\n';
}
cout << "sum: " << sum << '\n';
assert(sum == Fraction(10, 3));
cout << "ok\n\n";

cout << "testing operators * and / with "
      "sum = sum * oneThird / oneThird;\n";
sum = sum * oneThird / oneThird;
cout << "sum: " << sum << '\n';
assert(sum == Fraction(10, 3));
cout << "ok\n\n";

cout << "computing (1/2) + (-1/3) + (1/4) + (-1/5) +"
            " (1/6) + (-1/7) + (1/8) + (-1/9)" << "\n\n";
cout << setw(5) << 'k' << setw(15) << 'd' << " : " << 's' << '\n';
cout << setw(5) << '-' << setw(15) << '-' << " : " << '-' << '\n';
Fraction sum_alternate, sum_even, sum_odd;
double d = 0.0;
int one = 1;
for (int k = 2; k < 10; ++k)
{
  Fraction f3(1, k);
  if(k%2 == 0) sum_even += f3;
  else         sum_odd += f3;
```

```
   sum_alternate = sum_alternate + Fraction(one, k);
    d = d + static_cast<double>(one) / k;
    one = -one;
    cout << setw(5) << k << setw(15) << d << " : " << sum_alternate << '\n';

  }
  assert(sum_alternate == sum_even - sum_odd);
  cout << "ok\n\n";

  cout << "testing operator>> with cin >> f;\n";
  Fraction f;
  cin >> f;
  cout << "f : " << f << '\n';
  f = f + Fraction(6, 10) - f;

  cout << "\ntesting operator +=\n";
  f += 1;
  cout << "f : " << f << '\n';
  assert(f == Fraction(8, 5));
  cout << "f : " << f << '\n';
  cout << "ok\n\n";

  cout << "testing operator -=\n";
  f -= Fraction(1);
  cout << "f : " << f << '\n';
  assert(f == Fraction(3, 5));
  cout << "ok\n\n";

  cout << "testing operator *=\n";
  f *= Fraction(2, 7);
  cout << "f : " << f << '\n';
  assert(f == Fraction(6, 35));
  cout << "ok\n\n";

  cout << "testing operator /=\n";
  f /= Fraction(3, 5);
  cout << "f : " << f << '\n';
  assert(f == Fraction(2, 7));
  cout << "ok\n\n";

  cout << "testing Fraction + int\n";
  f = f + 1;
  cout << "f : " << f << '\n';
  assert(f == Fraction(9, 7));
  cout << "ok\n\n";
```

Test Driver Code: page 6/7

```

cout << "testing Fraction - int\n";
f = f - 1;
cout << "f : " << f << '\n';
assert(f == Fraction(2, 7));
cout << "ok\n\n";


cout << "testing int + Fraction\n";
f = 1 + f;
cout << "f : " << f << '\n';
assert(f == Fraction(9, 7));
cout << "ok\n\n";

cout << "testing int - Fraction\n";
f = 1 - f;
cout << "f : " << f << '\n';
assert(f == Fraction(-2, 7));
cout << "ok\n\n";

cout << "testing pointer behavior of f\n";
*f = Fraction(11, 22);
cout << "f : " << f << '\n';
assert(f == Fraction(1, 2));
cout << "ok\n\n";

cout << "testing string version of f\n";
cout << "f() : " << f() << '\n';
assert(f == Fraction(1, 2));
cout << "ok\n\n";

cout << "testing indexed Fraction objects\n";
f["top"] = 77; // same as nominator=77, but no normalization nor reduction
f["bottom"] = -154; // same as denominator=-154, but no normalization nor reduction
cout << "f : " << f << '\n';
assert(f["top"] == 77 && f["bottom"] == -154);
cout << "ok\n\n";

cout << "testing normalization\n";
f.normalize();
cout << "f : " << f << '\n';
assert(f["top"] == -77 && f["bottom"] == 154);
cout << "ok\n\n";
```

Test Driver Code: page 7/7

```

cout << "testing reduction to lowest form\n";
f.reduce();
cout << "f : " << f << '\n';
assert(f["top"] == -1 && f["bottom"] == 2);
cout << "ok\n\n";

cout << "Test completed successfully!" << endl;

return 0;
}
```

Test Drive Output

Output: page 1/4

```
Test Fraction and Fractional Computations
-----------------------------------------

testing default ctor with Fraction f0;
testing fraction == integer with f0 == 0
f0: 0
ok

testing 1-arg ctor with Fraction f1(5);
f1: 5

ok

testing copy ctor with Fraction f2 = f1;
f2: 5

ok

testing fraction == integer with f1 == 5;
ok

testing fraction == fraction with f1 == f2;
ok

testing fraction != fraction with !(f1 != f2);
ok

testing 2 args ctor with Fraction half = Fraction(1, 2);
half: 1/2

testing operator+ with f2 = f1 + half;
f2: 11/2 = 5 + 1/2

testing operator< with f1 < f2;
ok

testing operator<= with f1 <= f2;
ok

testing operator> with f2 > f1;
ok
```

Output: page 2/4

```

testing operator>= with f2 >= f1;
ok

testing operator!= with  f2 != f1;
ok

testing operator==, operator- with f1 == f2 - half;
ok

testing 2 args ctor with Fraction oneThird (1, 3);
oneThird: 1/3

testing assignment=, binary  +, -, and unary - with
f2 = f1 + oneThird - ( - oneThird );
f2: 17/3 = 5 + 2/3
ok

testing fractional expression with f2 = f1 - oneThird + ( - oneThird );
f2 *: 13/3 = 4 + 1/3
ok

testing post++ with f2 = f1++;
f1 : 6
f2 : 5
ok

testing pre++ with f2 = ++f1;
f1 : 7
f2 : 7
ok

testing post-- with f2 = f1--;
f1 : 6
f2 : 7
ok

testing pre-- with f2 = --f1;
f1 : 5
f2 : 5
ok

testing normalization with f1 == "6/10";
f1 : 3/5
ok
```

Output: page 3/4

```

computing sum = 1/3 + 1/3 + 1/3 + 1/3 + 1/3 + 1/3 + 1/3 + 1/3 + 1/3 + 1/3
sum: 1/3
sum: 2/3
sum: 1
sum: 4/3 = 1 + 1/3
sum: 5/3 = 1 + 2/3
sum: 2
sum: 7/3 = 2 + 1/3
sum: 8/3 = 2 + 2/3
sum: 3
sum: 10/3 = 3 + 1/3
sum: 10/3 = 3 + 1/3
ok

testing operators * and / with sum = sum * oneThird / oneThird;
sum: 10/3 = 3 + 1/3
ok

computing (1/2) + (-1/3) + (1/4) + (-1/5) + (1/6) + (-1/7) + (1/8) + (-1/9)

    k              d : s
    -              - : -
    2            0.5 : 1/2
    3       0.166667 : 1/6
    4       0.416667 : 5/12
    5       0.216667 : 13/60
    6       0.383333 : 23/60
    7       0.240476 : 101/420
    8       0.365476 : 307/840
    9       0.254365 : 641/2520
ok

testing operator>> with cin >> f;

numerator? -1
denumerator? -3
f : 1/3

testing operator +=
f : 8/5 = 1 + 3/5
f : 8/5 = 1 + 3/5
ok

testing operator -=
f : 3/5
ok
```

Output: page 4/4

```

testing operator *=
f : 6/35
ok

testing operator /=
f : 2/7
ok

testing Fraction + int
f : 9/7 = 1 + 2/7
ok

testing Fraction - int
f : 2/7
ok

testing int + Fraction
f : 9/7 = 1 + 2/7
ok

testing int - Fraction
f : -2/7
ok

testing pointer behavior of f
f : 1/2
ok

testing string version of f
f() : (1/2)
ok

testing indexed Fraction objects
f : 77/-154
ok

testing normalization
f : -77/154
ok

testing reduction to lowest form
f : -1/2
ok

Test completed successfully!
```

Marking scheme

60%	Program correctness
20%	OOP design, Proper use of C++ concepts and facilities, Error checking Simplicity and maintainability of implementation
10%	Format, clarity and completeness of output
10%	Concise documentation of nontrivial steps in code, choice of variable names, indentation and readability of program