Comp 5421/1-BB Assignment 3 Due 12 July 2014

## Objectives

1. To continue gaining experience with using the STL containers
2. To gain experience with using the STL adapters, such as **stack** and **queue**
3. To exploit the operator overloading facility (syntactic sugar) of the C++ language

## Your Task

In this assignment you will design an implement a class, named **Fraction**, that represents integer fractions together with a complete set of operations on fractions. Integer fractions are ratios of integers numbers, like $\frac{2}{3}$, $\frac{5}{1}$, $\frac{-3}{17}$, etc. See page 6 for complete specification of integer fractions in this assignment.

Your **Fraction** class should store the numerator and denominator of a fraction as two integers of type **long** and support the following operations:

- A constructor with optional parameters to allow the following constructions:

  `Fraction f1;` // `f1` $= \frac{0}{1}$

  `Fraction f2(−3);` // `f2` $= \frac{-3}{1}$ (*//also serves as conversion constructor*)

  `Fraction f3(14,−21);` // `f3` $= \frac{-2}{3}$

- A conversion constructor[1] with this prototype **Fraction(const string &)** to allow the following constructions:

  `string infix("1 + 1/2 + 1/3");`

  `Fraction f4(infix);` // `f4` $= \frac{11}{6}$

  `Fraction f5("1 + 7/2");` // `f5` $= \frac{9}{2}$

  The supplied string to this conversion constructor is expected to represent an infix[2] integer expression. With the help of the following two static fellow helpers

  ```
  static queue<string> Tokenize(const string & infixExpression);
  static Fraction evaluateInfix(queue<string> & infixQueue);
  ```

  this constructor initializes *this* **Fraction** to the fractional value of the input infix expression. More on this on the next page 4.

[1]A conversion constructor is a single-parameter constructor that is not declared **explicit**.
[2]See page 7 for a brief introduction of infix, postfix, and prefix notations.

- Accessor (getter) and mutator (setter) member functions for both the numerator and denominator of a **Fraction**. If invoked to set the denominator to zero, the mutator for denominator should throw an exception carrying the error message `string("Division by zero")`.

- A static member function with the following prototype to tokenize a specified infix integer expression into a queue of **string** tokens, where a token may represent an integer number, a parenthesis "(", ")", or any of the binary operators "+", "−", "∗", and "/".

```
static queue<string> Tokenize(const string & infixExpression);
```

- A static member function with the following prototype to evaluate and return the fractional value of an infix integer expression represented by a queue of **string** tokens.

```
static Fraction evaluateInfix(queue<string> & infixQueue);
```

- A **static** member function **int precedence(string op)** to compute and return the precedence of a given operator.

| Operator | Precedence |
|---|---|
| *, / | 2 |
| +, - | 1 |
| ( | 0 |

- A static member function **long gcd (long, long)** that computes and returns the greatest common divisor of the two supplied **long** values.

- A private member function **void normalize()** that normalizes *this* fraction. This member function must be called at the end of any operation that modifies a fraction; this is an important step, as normalization may slow down the rapid growth of the magnitudes of the numerator and denominator of fractions during the evaluation of some fractional expressions.

- Overload the following operators, providing the expected behaviors:

  ➢ Unary + and -. Non-members. Sample prototype:

  ```
  Fraction operator+ (const Fraction & rhs); // +f
  ```

  ➢ Binary +, −, * and /. Non-members. Sample prototype:

  ```
  Fraction operator+ (const Fraction& lhs, const Fraction& rhs); // A + B
  ```

  Notice that, with the help of the conversion constructor that created **f2** above, these operator overloads also allow symmetric operations with **int** operands, like **123 + f** and **f + 45**, for example.

  ➢ +=, −=, *= and /=. Members. Sample prototype:

  ```
  Fraction & Fraction::operator += (const Fraction & rhs); // A += B
  ```

  ➢ ==, <, !=, <=, >, >=. Non-members. Sample prototype:

  ```
  bool operator == (const Fraction &lhs, const Fraction & rhs); // A == B
  ```

Make sure that the operators **!=**, <=, >, >= are deduced through a combination of the logical operator **!** and the core relational operators == and <. Notice that these operator overloads also allow symmetric operations with **int** operands, like **123** < **f** and **f** >= **45**, for example. (how?)

- ++, −−. Members. Sample prototype:

```
Fraction & Fraction::operator++() // ++f
Fraction Fraction::operator++(int) // f++
```

- The function call operator() overload should take no parameters and return the string version of **this** Fraction:

```
string s4 = f4();
cout << s4; // print 11/6
```

- Insertion **operator**>>

```
cin >> f1; // read an infix integer expression, evaluate it, and store the result in f1
```

- Extraction **operator**<<

```
cout << f1; // print 0
cout << f2; // print −3
cout << f3; // print −2/3
cout << f4; // print 11/6 = 1 + 5/6
cout << f5; // print 9/2 = 4 + 1/2
```

## Fractional Expressions

Algebraic integer expressions in which every operand $x$ has been replaced by its fraction equivalent $\frac{x}{1}$ are called fractional expressions. For example, the integer expression

$$1 + (2 * 3)/5 - 4/7 \tag{1}$$

is equivalent to the fractional expression

$$\frac{1}{1} + \frac{\left(\frac{2}{1} \times \frac{3}{1}\right)}{\frac{5}{1}} - \frac{\frac{4}{1}}{\frac{7}{1}} \tag{2}$$

which is equivalent to the fractional value $\frac{22}{35}$.

## Evaluation of Infix Integer Expressions to Fractional Values

Recall that the classic approach to evaluate an infix integer expression involves two steps: (1) Convert the infix expression to its equivalent postfix expression, and (2) Evaluate the resulting postfix expression. The code for the classic approach is widely available on the Internet!

In this assignment, however, you must implement the following algorithm:

```
Algorithm:     EvaluateInfixExpression
Input:         infixQueue, a queue of strings storing the tokens in the input infix expression
               to be evaluated.
Output:        A Fraction representing the value of the input infix expression.
Precondition:  All operators in the input infix expression are binary operators.
Throws:        Nothing yet, but provides opportunities for you to detect errors and to throw
               exceptions if the input infix expression is not well formed (e.g., invalid operator,
               or missing an operator, operand, open or close parenthesis, etc.).

  1) Prepare an empty operand stack of type stack<Fraction>.
  2) Prepare an empty operator stack of type stack<string>.
  3) For each token in infixQueue
       a) If the token is a number then push it on the operand stack as a Fraction.
       b) Else if the token is an operator, then
            i) While the operator stack is not empty and the operator at the top has
               priority higher than or equal to the token then
                 A) Pop the top two operands from the operand stack
                 B) Pop the top operator from the operator stack
                 C) Apply the popped operator to the popped operands
                 D) Push the result onto the operand stack
           ii) Push the token onto the operator stack
       c) Else if the token is an open parenthesis, then push token onto the operator stack
       d) Else if the token is a close parenthesis, then
            i) While the operator stack is not empty and the operator at the top is not
               an open parenthesis
                 A) Pop the top two operands from the operand stack
                 B) Pop the top operator from the operator stack
                 C) Apply the popped operator to the popped operands
                 D) Push the result onto the operand stack
           ii) Pop the operator stack and discard the open parenthesis at the top
  4) While the operator stack is not empty
       a) Pop the top two operands from the operand stack
       b) Pop the top operator from the operator stack
       c) Apply the popped operator to the popped operands
       d) Push the result onto the operand stack
  5) Pop the operand stack and return it as the value of the input infix expression.
```

As an example, the following table traces the above algorithm applied to the infix integer expression $(9 - 1 + 2) * (8 - 3) + 6/2$:

| Current token | Unprocessed tokens | Operator Stack bottom ⇒ top | Operand Stack bottom ⇒ top |
|---|---|---|---|
| | (9 - 1 + 2)*(8 - 3) + 6/2 | | |
| ( | 9 - 1 + 2)*(8 - 3) + 6/2 | ( | |
| 9 | - 1 + 2)*(8 - 3) + 6/2 | ( | 9 |
| - | 1 + 2)*(8 - 3) + 6/2 | $(-$ | 9 |
| 1 | + 2)*(8 - 3) + 6/2 | $(-$ | 9 1 |
| + | 2)*(8 - 3) + 6/2 | (+ | 8 |
| 2 | )*(8 - 3) + 6/2 | (+ | 8 2 |
| ) | *(8 - 3) + 6/2 | | 10 |
| * | (8 - 3) + 6/2 | $*$ | 10 |
| ( | 8 - 3) + 6/2 | $*($ | 10 |
| 8 | - 3) + 6/2 | $*($ | 10 8 |
| - | 3) + 6/2 | $*(-$ | 10 8 |
| 3 | ) + 6/2 | $*(-$ | 10 8 3 |
| ) | + 6/2 | $*$ | 10 5 |
| + | 6/2 | + | 50 |
| 6 | /2 | + | 50 6 |
| / | 2 | +/ | 50 6 |
| 2 | | +/ | 50 6 2 |
| | | + | 50 3 |
| | | | 53 |

$$(9 - 1 + 2) * (8 - 3) + 6/2 = 53$$

## Suggestions

Start by writing the **Fraction** class without implementing the **EvaluateInfixExpression** algorithm. After you have tested and verified that your **Fraction** class operates correctly, proceed with the implementation of the **EvaluateInfixExpression** algorithm, Again test your code thoroughly. Your textbook has examples on overloading just about any operator.

## Integer Fractions

**Definition:** An integer fraction is a ratio of numbers of the form $\frac{a}{b}$, where $a$ and $b$ are both integers, with $b \neq 0$. The integers $a$ and $b$ are called the *numerator* and the *denominator* of the fraction $\frac{a}{b}$, respectively.

**Assumptions:** Without loss of generality, we assume that the denominator $b$ of any fraction $\frac{a}{b}$ is positive so that the sign of the fraction is the same as the sign of its numerator $a$. For example, $\frac{1}{-2}$ is expressed as $\frac{-1}{2}$, and $\frac{-1}{-2}$ as $\frac{+1}{2}$, or simply as $\frac{1}{2}$.

**Normalization:** A fraction is said to be in normalized form if its numerator and denominator have no common factors other than $\pm 1$. Thus, any fraction can be normalized by dividing both its numerator an denominator by their greatest common divisor ($gcd$). For example, since $5 = gcd(15, 20)$, the fraction $\frac{15}{20}$ normalizes to fraction $\frac{3}{4}$.

**Arithmetic operations:**

$$\begin{aligned}
\text{Addition:}\quad & \frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd} \\
\text{Subtraction:}\quad & \frac{a}{b} - \frac{c}{d} = \frac{ad-cb}{bd} \\
\text{Multiplication:}\quad & \frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd} \\
\text{Devision:}\quad & \frac{a}{b} / \frac{c}{d} = \frac{ad}{bc} \\
\text{Negation:}\quad & -\frac{a}{b} = \frac{-a}{b} \\
\text{Inverse:}\quad & \mathit{inverse}\left(\frac{a}{b}\right) = \frac{b}{a}, \quad a \neq 0 \\
\text{Normalization:}\quad & \mathit{normalize}\left(\frac{a}{b}\right) = \frac{a'}{b'}, \text{ where } a = ga',\ b = gb', \text{ and } g = gcd(a,b).
\end{aligned}$$

**Relational operations** The the following simple definitions are based on the assumption that both denominators $b$ and $d$ have the same sign; in this assignment they are both assumed to be positive.

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

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

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

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

## Representaion of Arithmetic Expressions

Arithmetic expressions may be written in a variety of notations depending on where an operator (such a $+$, -, $*$, etc.) is placed relative to its operands (numbers and variables):

Prefix: Operator *before* the operands: $+\ 2\ 3$

Infix: Operator *between* the operands: $(2\ +\ 3)$ or $2\ +\ 3$

Postfix: Operator *after* the operands: $2\ 3\ +$

To avoid ambiguity, the *infix* notation requires the use of parentheses as well as operator associativity and precedence rules. For example, if the operators $*$ has higher precedence than the operator $+$, then the expression "*2 times the sum of 3 and 4*" is written as follows:

Prefix: $*\ 2\ +\ 3\ 4$

Infix: $2\ *\ (\ 3\ +\ 4\ )$ Must use '(' and ')' to avoid ambiguity here

Postfix: $2\ 3\ 4\ +\ *$

Notice that in all three notations, the operands appear in the same order, but the operators do not.

Unlike infix expressions, however, *prefix* and *postfix* expressions *never* require the use of cumbersome parentheses, precedence rules, or rules for association, and hence are convenient for programmers.

The algorithms for converting to and evaluating postfix and prefix expressions have similar complexity, but postfix expressions are somewhat simpler to evaluate on computers.

## Marking scheme

| | |
|---|---|
| 60% | Program correctness. |
| 20% | Implementation of algorithm EvaluateInfixExpression |
| 10% | Proper use of C++ concepts and the STL |
| 10% | Concise documentation of nontrivial steps in code, choice of variable names, indentation and readability of program |

## Test Driver Code

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

int main()
{
   cout << "Test Fraction and Fractional Computation" << '\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';
   assert(f0 == 0); // test fraction == integer
   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";

   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";

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

cout << "testing fractional expression with f2 = f1 - oneThird + ( - oneThird );" <<
f2 = f1 - oneThird + (-oneThird); // assignment=, binary +, -, and 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";

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";

cout << "testing conversion constructor with f1 = \"3/5\";" << '\n';
cout << "testing infix evaluation with f1 = \"3/5\";" << '\n';
f1 = "3/5";
cout << "f1 : " << f1 << '\n';
assert(f1 == Fraction(3, 5));
cout << "ok\n\n";

cout << "testing normalization with f1 == \"6/10\";" << '\n';
assert(f1 == "6/10");
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 f3 = 1/2 + 1/3 + 1/4 + 1/5 + 1/6 + 1/7 + 1/8 + 1/9 + 1/10;" << '\n
Fraction f3;
f3 = "1/2 + 1/3 + 1/4 + 1/5 + 1/6 + 1/7 + 1/8 + 1/9 + 1/10";
cout << "f3: " << f3 << '\n';
assert(Fraction("1/2 + 1/4 + 1/6 + 1/8 + 1/10")
   +
   Fraction("1/3 + 1/5 + 1/7 + 1/9")
   == f3);
cout << "ok\n\n";

cout << "computing (1/2) + (-1/3) + (1/4) + (-1/5) + (1/6) + (-1/7) + (1/8) + (-1/9)"
cout << setw(5) << 'k' << setw(15) << 'd' << " : " << 's' << '\n';
cout << setw(5) << '-' << setw(15) << '-' << " : " << '-' << '\n';
Fraction s;
double d = 0.0;
int one = 1;
for (int k = 2; k < 10; ++k)
{

   d = d + static_cast<double>(one) / k;
   s = s + Fraction(one, k);
   one = -one;
   cout << setw(5) << k << setw(15) << d << " : " << s << '\n';
}
assert(Fraction("1/2 + 1/4 + 1/6 + 1/8")
   -
   Fraction("1/3 + 1/5 + 1/7 + 1/9")
   == s);
```

```
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 << "testing 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";

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 << "Test over successfully!" << endl;

    return 0;
}
```

## Test Driver Output

```
Test Fraction and Fractional Computation
----------------------------------------

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

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 conversion constructor with f1 = "3/5";
testing infix evaluation with f1 = "3/5";
f1 : 3/5
ok
```

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

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 f3 =  1/2 + 1/3 + 1/4 + 1/5 + 1/6 + 1/7 + 1/8 + 1/9 + 1/10;
f3: 4861/2520 = 1 + 2341/2520
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? 7
denumerator? 123
f : 7/123
testing operator +=
f : 8/5 = 1 + 3/5
f : 8/5 = 1 + 3/5
ok

testing operator -=
f : 3/5
ok

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

Test over successfully!
Press any key to continue . . .
```