Indian Institute of Technology, Kharagpur

Department of Computer Science and Engineering

Assignment 3: C++ Programming, Spring 2013-14

Software Engineering (CS 29006)

Assignment Date:	07-Feb-2014	**Submission Deadline:**	23:55 hrs, 14-Feb-2014
Revised Assignment Date:	13-Feb-2014	**Revised Submission Deadline:**	23:55 hrs, 16-Feb-2014

Instructions

- All assignments in this set should be coded in C/C++.
- Zero marks for a submission if it does not pass the plagiarism test or if you copied from someone in the class.
- Zero marks for a submission and 20% deduction from all previous assignments if someone in the class copied from you.

Note: *Problems completed in the earlier version of Assignment 3 will be acceptable. But the credit earned will be as par this revised version only. Extra parts will not be evaluated.*

---

1. You need to develop a Fraction Data type in C++ to deal with fractional (rational) numbers. A rational number `r` is represented as a fraction `p/q` where `p` and `q` are integers, `q>0` and `p` and `q` are mutually prime (`gcd(p, q) = 1`). These numbers should support the operations for a Fraction type as defined in the following class definition (`Fraction.hxx`):

```
#include <iostream>                // Defines istream & ostream for IO
using namespace std;

class Fraction {

public:
    // CONSTRUCTORS
    Fraction(int = 1, int = 1);    // Uses default parameters.
    explicit Fraction(double);     // explicit double to Fraction conversion
    Fraction(const Fraction&);     // Copy Constructor

    // DESTRUCTOR
    ~Fraction();                   // No virtual destructor needed

    // BASIC ASSIGNMENT OPERATOR
    Fraction& operator=(const Fraction&);

    // UNARY ARITHMETIC OPERATORS
    Fraction operator-();          // Operand 'this' implicit
    Fraction operator+();
    Fraction& operator--();        // Pre-decrement. Dividendo. p/q <-- p/q - 1
    Fraction& operator++();        // Pre-increment. Componendo. p/q <-- p/q + 1
    Fraction operator--(int);      // Post-decrement. Lazy Dividendo. p/q <-- p/q - 1. Returns old p/q
    Fraction operator++(int);      // Post-increment. Lazy Componendo. p/q <-- p/q + 1. Returns old p/q

    // BINARY ARITHMETIC OPERATORS USING FRIEND FUNCTIONS
    friend Fraction operator+(const Fraction&, const Fraction&);
    friend Fraction operator-(const Fraction&, const Fraction&);
    friend Fraction operator*(const Fraction&, const Fraction&);
    friend Fraction operator/(const Fraction&, const Fraction&);
    friend Fraction operator%(const Fraction&, const Fraction&);
    // BINARY RELATIONAL OPERATORS
    bool operator==(const Fraction&);
    bool operator!=(const Fraction&);
    bool operator<(const Fraction&);
    bool operator<=(const Fraction&);
    bool operator>(const Fraction&);
    bool operator>=(const Fraction&);

    // ADVANCED ASSIGNEMENT OPERATORS
    Fraction& operator+=(const Fraction&);
    Fraction& operator-=(const Fraction&);
    Fraction& operator*=(const Fraction&);
    Fraction& operator/=(const Fraction&);
    Fraction& operator%=(const Fraction&);

    // SPECIAL OPERATORS
    Fraction operator!();          // Inverts a fraction. !(p/q) = q/p

    // BASIC I/O using FRIEND FUNCTIONS
    friend ostream& operator<<(ostream&, const Fraction&);
    friend istream& operator>>(istream&, Fraction&);

    // CONSTANTS OF DATATYPE
    static const Fraction    sc_fUnity;   // Defines 1/1
    static const Fraction    sc_fZero;    // Defines 0/1

    // STATIC UTILITY FUNCTIONS
    static const int precision()     { return 1000000; };
    static int gcd(int, int);      // Finds the gcd for two +ve integers
    static int lcm(int, int);      // Finds the lcm for two +ve integers

protected:
    // COMPONENT FUNCTIONS
    int GetNumerator() { return iNumerator_; }
    unsigned int GetDenominator() { return uiDenominator_; }

private:
    // DATA MEMBERS
    int             iNumerator_;       // The Numerator
    unsigned int    uiDenominator_;    // The Denominator

    // OTHER METHOD MEMBERS
    Fraction& Normalize();             // Normalizes a fraction
};
```

(a) Implement this class. [**20 Marks**]

Grading Guideline:

Completeness & Correctness	*70%*
Code Quality (simplicity, readability, efficiency, reusability, standard library usage, robustness)	*20%*
Comments	*10%*

(b) Your class will be tested by the TA using the following function (`TestFraction.cxx`):

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

#include "Fraction.hxx"

void TestFraction()
{
    cout << "\nTest Fraction Data Type" << endl;
```

```
// CONSTRUCTORS
// ------------
Fraction f1(5, 3);
Fraction f2(7.2);
Fraction f3;

cout << "Fraction f1(5, 3) = " << f1 << endl;
cout << "Fraction f2(7.2) = " << f2 << endl;
cout << "Fraction f3 = " << f3 << endl;

// BASIC ASSIGNMENT OPERATOR
// -------------------------
// Fraction& operator=(const Fraction&);
cout << "Assignment (Before): f3 = " << f3 << ". f1 = " << f1 << endl;
f3 = f1;
cout << "Assignment (After): f3 = " << f3 << ". f1 = " << f1 << endl;

f3 = Fraction::sc_fUnity;

// UNARY ARITHMETIC OPERATORS
// --------------------------
// Fraction operator-();        // Operand 'this' implicit
f3 = -f1;
cout << "Unary Minus: f3 = " << f3 << ". f1 = " << f1 << endl;

// Fraction operator+();

// Fraction operator--();        // Pre-decrement. Dividendo
f3 = Fraction::sc_fUnity;
cout << "Pre-Decrement (Before): f3 = " << f3 << ". f1 = " << f1 << endl;
f3 = --f1;
cout << "Pre-Decrement (After): f3 = " << f3 << ". f1 = " << f1 << endl;

// Fraction operator--(int);    // Post-decrement. Lazy Dividendo
f3 = Fraction::sc_fUnity;
cout << "Post-Decrement (Before): f3 = " << f3 << ". f1 = " << f1 << endl;
f3 = f1--;
cout << "Post-Decrement (After): f3 = " << f3 << ". f1 = " << f1 << endl;

// Fraction operator++();        // Pre-increment. Componendo
f3 = Fraction::sc_fUnity;
cout << "Pre-Increment (Before): f3 = " << f3 << ". f1 = " << f1 << endl;
f3 = ++f1;
cout << "Pre-Increment (After): f3 = " << f3 << ". f1 = " << f1 << endl;

// Fraction operator++(int);    // Post-increment. Lazy Componendo
f3 = Fraction::sc_fUnity;
cout << "Post-Increment (Before): f3 = " << f3 << ". f1 = " << f1 << endl;
f3 = f1++;
cout << "Post-Increment (After): f3 = " << f3 << ". f1 = " << f1 << endl;

// BINARY ARITHMETIC OPERATORS USING FRIEND FUNCTIONS
// --------------------------------------------------
// friend Fraction operator+(const Fraction&, const Fraction&);
f1 = Fraction(5, 12);
f2 = Fraction(7, 18);
f3 = f1 + f2;
cout << "Binary Plus: f3 = " << f3 << ". f1 = " << f1 << ". f2 = " << f2 << endl;

// friend Fraction operator-(const Fraction&, const Fraction&);
f1 = Fraction(16, 3);
f2 = Fraction(22, 13);
f3 = f1 - f2;
cout << "Binary Minus: f3 = " << f3 << ". f1 = " << f1 << ". f2 = " << f2 << endl;

// friend Fraction operator*(const Fraction&, const Fraction&);
```

```
        f1 = Fraction(5, 12);
        f2 = Fraction(18, 25);
        f3 = f1 * f2;
        cout << "Multiply: f3 = " << f3 << ". f1 = " << f1 << ". f2 = " << f2 << endl;

        // friend Fraction operator/(const Fraction&, const Fraction&);
        f1 = Fraction(5, 12);
        f2 = Fraction(7, 18);
        f3 = f1 / f2;
        cout << "Divide: f3 = " << f3 << ". f1 = " << f1 << ". f2 = " << f2 << endl;

        // friend Fraction operator%(const Fraction&, const Fraction&);
        f1 = Fraction(5, 12);
        f2 = Fraction(7, 18);
        f3 = f1 % f2;
        cout << "Residue: f3 = " << f3 << ". f1 = " << f1 << ". f2 = " << f2 << endl;

        // BINARY RELATIONAL OPERATORS
        // ---------------------------
        // bool operator==(const Fraction&);
        f1 = Fraction(5, 12);
        f2 = Fraction(7, 18);
        bool bTest = f1 == f2;
        cout << "Equal: Test = " << ((bTest)? "true": "false") <<
            ". f1 = " << f1 << ". f2 = " << f2 << endl;

        // bool operator!=(const Fraction&);
        bTest = f1 != f2;
        cout << "Not Equal: Test = " << ((bTest)? "true": "false") <<
            ". f1 = " << f1 << ". f2 = " << f2 << endl;

        // bool operator<(const Fraction&);
        bTest = f1 < f2;
        cout << "Less: Test = " << ((bTest)? "true": "false") <<
            ". f1 = " << f1 << ". f2 = " << f2 << endl;

        // bool operator<=(const Fraction&);
        f1 = Fraction(5, 12);
        f2 = Fraction(7, 18);
        f3 = Fraction(5, 12);
        bTest = f1 <= f2;
        cout << "Less Equal: Test = " << ((bTest)? "true": "false") <<
            ". f1 = " << f1 << ". f2 = " << f2 << endl;
        bTest = f1 <= f3;
        cout << "Less Equal: Test = " << ((bTest)? "true": "false") <<
            ". f1 = " << f1 << ". f3 = " << f3 << endl;

        // bool operator>(const Fraction&);
        bTest = f1 > f2;
        cout << "Greater: Test = " << ((bTest)? "true": "false") <<
            ". f1 = " << f1 << ". f2 = " << f2 << endl;

        // bool operator>=(const Fraction&);
        bTest = f1 >= f2;
        cout << "Greater Equal: Test = " << ((bTest)? "true": "false") <<
            ". f1 = " << f1 << ". f2 = " << f2 << endl;
        bTest = f1 >= f3;
        cout << "Greater Equal: Test = " << ((bTest)? "true": "false") <<
            ". f1 = " << f1 << ". f3 = " << f3 << endl;

        // ADVANCED ASSIGNEMENT OPERATORS
        // ------------------------------
        // Fraction& operator+=(const Fraction&);
        f1 = Fraction(5, 12);
        f2 = Fraction(7, 18);
        f3 = f2;
```

```
    f2 += f1;
    cout << "+=: f2 = " << f2 << ". f1 = " << f1 << ". f2 (before) = " << f3 << endl;
    f3 = f2;
    f2 += f2;
    cout << "+=: f2 = " << f2 << ". f2 (before) = " << f3 << endl;

    // Fraction& operator-=(const Fraction&);
    f1 = Fraction(5, 12);
    f2 = Fraction(7, 18);
    f3 = f2;
    f2 -= f1;
    cout << "-=: f2 = " << f2 << ". f1 = " << f1 << ". f2 (before) = " << f3 << endl;
    f3 = f2;
    f2 -= f2;
    cout << "-=: f2 = " << f2 << ". f2 (before) = " << f3 << endl;

    // Fraction& operator*=(const Fraction&);
    f1 = Fraction(5, 12);
    f2 = Fraction(7, 18);
    f3 = f2;
    f2 *= f1;
    cout << "*=: f2 = " << f2 << ". f1 = " << f1 << ". f2 (before) = " << f3 << endl;
    f3 = f2;
    f2 *= f2;
    cout << "*=: f2 = " << f2 << ". f2 (before) = " << f3 << endl;

    // Fraction& operator/=(const Fraction&);
    f1 = Fraction(5, 12);
    f2 = Fraction(7, 18);
    f3 = f2;
    f2 /= f1;
    cout << "/=: f2 = " << f2 << ". f1 = " << f1 << ". f2 (before) = " << f3 << endl;
    f3 = f2;
    f2 /= f2;
    cout << "/=: f2 = " << f2 << ". f2 (before) = " << f3 << endl;

    // Fraction& operator%=(const Fraction&);
    f1 = Fraction(7, 18);
    f2 = Fraction(5, 12);
    f3 = f2;
    f2 %= f1;
    cout << "%=: f2 = " << f2 << ". f1 = " << f1 << ". f2 (before) = " << f3 << endl;
    f3 = f2;
    f2 %= f2;
    cout << "%=: f2 = " << f2 << ". f2 (before) = " << f3 << endl;

    return;
}
```

Hence compliance to this function is critical. [**20 Marks**]

Grading Guideline: Based on percentage of tests passed / failed.

(c) Build a Rational number calculator (with console-based text interface) using the type developed by you. [**10 Marks**]

Grading Guideline:

Completeness of Design	*20%*
Completeness & Correctness of Implementation	*40%*
Completeness of Tests	*20%*
Code Quality (simplicity, readability, efficiency, reusability, standard library usage, robustness)	*10%*
Comments	*10%*

2. You need to develop Poly Data type in C++ to deal with polynomials of value type `Fraction` and coefficient type `int`. These polynomials should support the operations for a Poly type as defined in the following class definition (`Polynomial.hxx`):

```
#include <iostream>// Defines istream & ostream for IO
#include <vector>
using namespace std;

class Poly {

public:
    // CONSTRUCTORS
    Poly(unsigned int = 0);    // Uses default parameters.
    Poly(const Poly&);         // Copy Constructor

    // DESTRUCTOR
    ~Poly() {}                 // No virtual destructor needed

    // BASIC ASSIGNMENT OPERATOR
    Poly& operator=(const Poly&);

    // UNARY ARITHMETIC OPERATORS
    Poly operator-();           // Operand ’this’ implicit
    Poly operator+();

    // BINARY ARITHMETIC OPERATORS
    Poly operator+(const Poly&);
    Poly operator-(const Poly&);

    // ADVANCED ASSIGNMENT OPERATORS
    Poly& operator+=(const Poly&);
    Poly& operator-=(const Poly&);

    // BASIC I/O using FRIEND FUNCTIONS
    friend ostream& operator<<(ostream& os, const Poly& p);

    friend istream& operator>>(istream& is, Poly& p);

    // METHODS
    Fraction Evaluate(const Fraction&); // Evaluates the polynomial - use Horner’s Rule

private:

    // DATA MEMBERS
    unsigned int    degree_;
    vector<int>     coefficients_;
};
```

(a) Implement this class. **[20 Marks]**

Grading Guideline:

Completeness & Correctness	*70%*
Code Quality (simplicity, readability, efficiency, reusability, standard library usage, robustness)	*20%*
Comments	*10%*

(b) Your class will be tested by the TA using the following function (`TestPoly.cxx`):

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

#include "Fraction.hxx"
#include "Polynomial.hxx"

void TestPoly()
{
    cout << "\nTest Poly Data Type" << endl;
    // Polynomial with int value and int coefficients
    Poly p(10);

    cout << "Input Poly: p(x)" << endl;
    cin >> p;
    cout << "\np(x) = " << p << endl;

    Fraction f;
    cout << "Input Fraction" << endl;
    cin >> f;
    cout << "p(" << f << ") = " << p.Evaluate(x) << endl;

    Poly q = p;
    cout << "Copied Polynomial: " << q << endl;

    Poly r;
    r = p;
    cout << "Assigned Polynomial: " << r << endl;

    r = -p;
    cout << "Negated Polynomial -p(x) = " << r << endl;

    cout << "Input Poly<int, int>: q(x)" << endl;
    cin >> q;
    cout << "\nq(x) = " << q << endl;

    r = p + q;
    cout << "p(x) + q(x) = " << r << endl;

    r = p - q;
    cout << "p(x) - q(x) = " << r << endl;

    p += q;
    cout << "p(x) <-- p(x) + q(x): " << p << endl;

    q -= p;
    cout << "q(x) <-- q(x) - p(x): " << q << endl;

    return;
}
```

Hence compliance to this function is critical. **[20 Marks]**

Grading Guideline: Based on percentage of tests passed / failed.

3. This problem tests your understanding of implementing a data structure in C++.

   (a) Design and implement a stack of integers (int) in C. Use your stack to convert an infix expression with integer constants to postfix and evaluate the expression. Assume the operators +, -, * and / in your expression.

   For example, if the infix expression is 2+3*4 then the postfix expression is 234*+ and the evaluated value is 14.

   Handle all corner cases in your code. The container for your stack should be dynamically allocated as a linked list.

   (b) Repeat (a) in C++. For the stack of int, you should implement a Stack class for underlying int element types.

[15+15 = 30 Marks]

Grading Guideline:

Completeness of Design	*20%*
Completeness & Correctness of Implementation	*40%*
Completeness of Tests	*20%*
Code Quality (simplicity, readability, efficiency, reusability, standard library usage, robustness)	*10%*
Comments	*10%*