Computer Science & Engineering Department
I. I. T. Kharagpur

Software Engineering: CS20006

Assignment – 2: Class & UDT *Marks: 50*

Assign Date: *20^{th} January, 2021* Submit Date: *23:55, 1^{st} February, 2021*

We want to design a User-Defined Datatype (UDT) `Fraction` for rational numbers. It should behave like the built-in numerical types (for example, `int`). The concept, interface and implementation of the datatype are discussed below before the engineering requirements and tasks are specified.

1. **Concept**: Here we outline the basic concept for a `Fraction` datatype. `class Fraction` is to be designed as a UDT representing reduced (normalized) proper fractions (rational numbers) of the form $\frac{p}{q}$ where $p, q \in$ `int`, $q > 0$, and $gcd(p, q) = 1$. Hence, $\frac{12}{18}$, $\frac{6}{9}$, *and* $\frac{2}{3}$ all will be represented only as $\frac{2}{3}$.

   Most operations available in C++ for `int` type should be available in their respective syntax and semantics for `Fraction`. Some operations (like variants of assignments) have been skipped for simplicity and some operations (like logical negation) have been given new semantics.

2. **Interface**: The expected behavior of the datatype (operations) are discussed in this section.

   - **Constructors**: All constructors must only create normalized fractions. They include
     - Constructor with a pair of `int` values for the numerator and denominator properly defaulted.
       * Invoked as a default constructor, it would create: $\frac{p}{q} \equiv \frac{1}{1}$
       * Invoked with one parameter m, it would create: $\frac{p}{q} \equiv \frac{m}{1}$
       * Invoked with two parameters m and n, it would create: $\frac{p}{q} \equiv \frac{m}{n}$. If $n < 0$, the construction will be done with $-m$ and $-n$. If $m = 0$, n is taken to be 1. Naturally, if $n = 0$, the program would call `exit(1)`[1].
     - Constructor from a single `double` value d. The class `Fraction` should use a preset `int` *precision* to convert from d as:
       $$\frac{p}{q} \equiv \frac{\lfloor d * precision \rfloor}{precision}$$
       This constructor should not allow implicit conversion.
     - Copy Constructor: Should implement the usual semantics
   - **Destructor**: Should implement the usual semantics
   - **Copy Assignment Operator**: Should implement the usual semantics
   - **Unary Arithmetic Operators**: The operator should take a `Fraction` value and produce a `Fraction` value after the operation. It may or may not change the operand.
     - Unary minus operator: $-\left(\frac{p}{q}\right) \equiv \frac{-p}{q}$
     - Unary plus operator: $+\left(\frac{p}{q}\right) \equiv \frac{+p}{q}$
     - Prefix pre-decrement operator for dividendo returns $\frac{p}{q} \leftarrow \frac{p}{q} - 1 = \frac{p-q}{q}$
     - Prefix pre-increment operator for componendo returns $\frac{p}{q} \leftarrow \frac{p}{q} + 1 = \frac{p+q}{q}$
     - Postfix post-decrement operator for lazy dividendo returns $\frac{p}{q}$ & performs $\frac{p}{q} \leftarrow \frac{p}{q} - 1 = \frac{p-q}{q}$
     - Postfix post-increment operator for lazy componendo returns $\frac{p}{q}$ & performs $\frac{p}{q} \leftarrow \frac{p}{q} + 1 = \frac{p+q}{q}$
   - **Binary Arithmetic Operators** (using `friend` functions): The operator should take a pair of `Fraction` values and produce a `Fraction` value after the operation. The operands must not be modified. The operator returns the result of:
     - *Addition*: Adding the First operand to the Second operand
     - *Subtraction*: Subtracting the Second operand from the First operand
     - *Multiplication*: Multiplying the First operand with the Second operand

[1] Constructors should not throw an exception

- *Division*: Dividing the First operand by the Second operand. Should throw an exception if the divider (Second operand) is zero
- *Remainder*: Remainder[2] of division of the First operand by the Second operand. Should throw an exception if the divider (Second operand) is zero

- **Binary Relational Operators**: The operator should take a pair of `Fraction` values and produce a `bool` result. The operands must not be modified. The operator returns `true` when
  - Operands are *Equal*
  - Operands are *Not Equal*
  - First operand is *Less than* Second operand
  - First operand is *Less than or Equal to* Second operand
  - First operand is *More than* Second operand
  - First operand is *More than or Equal to* Second operand
- **Special Operators**: The negation (!) operator should take a `Fraction` value and produce an inverted `Fraction` value as $!\left(\frac{p}{q}\right) \equiv \frac{q}{p}$. The operand must not be modified and should throw an exception if the fraction is zero.
- **I/O Operators** (using `friend` functions): Should implement the usual semantics for streaming.
  - *Output*: Streaming to `ostream`. $\frac{p}{q}$ to be output as `p / q`. The divider should be omitted if $\frac{p}{q}$ is a whole number.
  - *Input*: Streaming from `istream`. Read a pair of `int` p and q, form $\frac{p}{q}$ and normalize
- **Constants of the Datatype**: Unity and Zero should be defined as static constants
  - `sc_fUnity` = `Fraction()` $\equiv \frac{1}{1}$
  - `sc_fZero` = `Fraction(0)` $\equiv \frac{0}{1}$
- **Utility Functions of the Datatype**: The following static functions should be available:
  - `precision()` to return the constant to be used in conversion from `double`
  - `int gcd(int, int)` finds the greatest common divisor (gcd) for two positive integers
  - `int lcm(int, int)` finds the least common multiple (lcm) for two positive integers

3. **Implementation**: A suggestive implementation for the datatype is given here. Any other choice of implementation consistent with the interfaces would be acceptable.

   - **Data Members**
     - Numerator of type `int`
     - Denominator of type `unsigned int`
   - **Utility Function**
     - A function to normalize a fraction to its proper form. Also, sets q to 1 if p is 0.

4. **Engineering**: Finally, we engineer the concept based on the expectations of the interface and implementation to create the `Fraction` datatype. The tasks therein include:

   (a) Design the interface for `class Fraction` containing the exposed member functions (signatures only) as specified in **Interface** section. Explain the design considerations (like parameters, types, call/return-by-value/reference, const-ness, initialization, exception etc.) in comments before every member function to illustrate your design understanding and depth.

   Put the design in `"Fraction.hxx"`

   Note: Marks will be distributed as: **[20]**

[2] What is the meaning of remainder in rational algebra?

Constructors	**[0.5 + 1 + 0.5 = 2]**
Destructor	**[0.5]**
Copy Assignment Operator	**[0.5]**
Unary Arithmetic Operators	**[0.5 * 6 = 3]**
Binary Arithmetic Operators	**[1 * 5 = 5]**
Binary Relational Operators	**[0.5 * 6 = 3]**
Special Operators	**[0.5]**
I/O Operators	**[1 * 2 = 2]**
Constants of the Datatype	**[1 * 2 = 2]**
Utility Functions of the Datatype	**[0.5 * 3 = 1.5]**

(b) Add the implementation for `class Fraction` containing the concealed data members and member functions as specified in **Implementation** section. Explain the design considerations in comments before every member. **[0]**

(c) Implement all member functions of `class Fraction` and `inline` as appropriate. Add implementation notes as comments for every function.

Put the implementations of the `inline` member functions in `"Fraction.hxx"`. Put the non-`inline` member function implementations and static constants in `"Fraction.cxx"`

Note: Marks will be distributed as: **[30]**

Constructors	**[2 * 3 = 6]**
Destructor	**[1]**
Copy Assignment Operator	**[2]**
Unary Arithmetic Operators	**[0.5 * 4 + 1 * 2 = 4]**
Binary Arithmetic Operators	**[1 * 5 = 5]**
Binary Relational Operators	**[0.5 * 6 = 3]**
Special Operators	**[1 + 1 = 2]**
I/O Operators	**[1.5 * 2 = 3]**
Constants of the Datatype	**[0.5 * 2 = 1]**
Utility Functions of the Datatype	**[3]**

(d) Test your implementation using the `void TestFraction()` below from `"TestFraction.cxx"` file. Finally, `int main()` (file `"Main.cxx"`) invokes `TestFraction()` to perform the test.

Put your test results in `"Fraction.out"`

Note:

- If you are unable to write the interface (signature) of a member function, skip it in `"Fraction.hxx"` and comment out its use in `"TestFraction.cxx"`
- If you are unable to write the implementation of a member function, skip it in `"Fraction.hxx"` or `"Fraction.cxx"` and comment out its use in `"TestFraction.cxx"`
- Obviously, the above exclusions will not work if you fail to do the basic methods like constructor, destructor, assignment etc. In that case, please contact the TA.

(e) Submit `"Fraction.hxx"`, `"Fraction.cxx"`, `"TestFraction.cxx"`, `"Main.cxx"`, and `"Fraction.out"` in a zip file assgn-2-<your roll>.zip.

Note:

- `"Fraction.hxx"` and `"Fraction.cxx"` will be evaluated
- `"Fraction.out"` will be checked for completeness of operators
- `"TestFraction.cxx"` and `"Main.cxx"` will be used if the TA needs to build and rerun the entire project
- If a particular operator does not compile, its marks for interface design will be zero
- If a particular operator does not give correct result by test, its marks for implementation will be zero

```
// File: TestFraction.cxx
// Contains: void TestFraction()

/************ C++ Headers **********************************/
#include <iostream>
using namespace std;

/************ PROJECT Headers ******************************/
#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 ASSIGNEMENT OPERATOR
    // --------------------------
    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
    // --------------------------
    f3 = -f1;
    cout << "Unary Minus: f3 = " << f3 << ". f1 = " << f1 << endl;

    // 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;

    // 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;

    // 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;

    // 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
// --------------------------------------------------
f1 = Fraction(5, 12);
f2 = Fraction(7, 18);
f3 = f1 + f2;
cout << "Binary Plus: f3 = " << f3 << ". f1 = " << f1
     << ". f2 = " << f2 << endl;

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

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

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

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

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

bTest = f1 != f2;
cout << "Not Equal: Test = " << ((bTest)? "true": "false")
     << ". f1 = " << f1 << ". f2 = " << f2 << endl;

bTest = f1 < f2;
cout << "Less: Test = " << ((bTest)? "true": "false")
     << ". f1 = " << f1 << ". f2 = " << f2 << endl;

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;
```

```
    bTest = f1 > f2;
    cout << "Greater: Test = " << ((bTest)? "true": "false")
         << ". f1 = " << f1 << ". f2 = " << f2 << endl;

    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;

    return;
}

// End-of-File: TestFraction.cxx



// File: Main.cxx
// Contains: int main()

/************ C++ Headers ***********************************/
#include <iostream>
using namespace std;

/************ PROJECT Headers *******************************/
#include "Fraction.hxx"

void TestFraction();

int main() {
    TestFraction();

    return 0;
}

// End-of-File: Main.cxx
```