MEMBER OPERATOR OVERLOADS

Workshop 5 (10 marks – 3% of your final grade)

In this workshop, you are to overload member operators for a class type.

LEARNING OUTCOMES

Upon successful completion of this workshop, you will have demonstrated the abilities

- to overload an operator as a member function of a class type
- to access the current object from within a member function
- to identify the lifetime of an object, including a temporary object
- to describe to your instructor what you have learned in completing this workshop

SUBMISSION POLICY

The "in-lab" section is to be completed during your assigned lab section. It is to be completed and submitted by the end of the workshop period. If you attend the lab period and cannot complete the in-lab portion of the workshop during that period, ask your instructor for permission to complete the in-lab portion after the period. If you do not attend the workshop, you can submit the "in-lab" section along with your "at-home" section (with a penalty; see below). The "at-home" portion of the lab is due on the day that is two days before your next scheduled workshop (23:59:59).

All your work (all the files you create or modify) must contain your name, Seneca email and student number.

You are responsible to back up your work regularly.

LATE SUBMISSION PENALTIES:

- *In-lab* portion submitted late, with *at-home* portion: **0** for *in-lab*. Maximum of 7/10 for the entire workshop.

- If any of *in-lab*, *at-home* or *reflection* portions is missing, the mark for the workshop will be **0**/10.

IN-LAB (30%)

In this workshop, you are to design a class that represents fractions. Fractions are common in many domains. For example:

- in baking, recipe ingredients are often listed in fractional measures (1/2 cup of flour for 1 batch of cookie dough),
- in many commercials, retailers use statistics expressed in fractional form to get you to buy their products:
  - 4/5 dentists approve this toothpaste
  - 9/10 women prefer this lipstick

REVIEW OF FRACTIONS

A fraction is a number that can be represented as the ratio of one integer (numerator) over another integer (denominator). For example, 4/5, 3/7 and 12/2 are fractions; the 4, 3, and 12 are numerators and the 5, 7 and 2 are denominators. By convention, the denominator is a positive integer. A fraction with a denominator of 0 is ill-defined.

The following examples show the results of two basic operations on two fractions (a/b) and (c/d):

Addition:

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

Multiplication:

$$\frac{a}{b} * \frac{c}{d} = \frac{ac}{bd}$$

YOUR TASK

Design and code a `Fraction` class that holds the information for a single fraction and defines the set of admissible mathematical operations on that fraction. The main function will use your design to perform these operations as if a `Fraction` object is just another C++ built-in arithmetic type.

Store your class definition in the header file named `Fraction.h` and your member function definitions in the implementation file named `Fraction.cpp`.

Your design includes the following member functions:

default constructor (a constructor with no parameters): this constructor sets the object to a safe empty state;

constructor with 2 parameters: receives the numerator and denominator of a fraction and stores the data only if it is valid. The data is valid if the numerator is non-negative-valued and the denominator is positive-valued. If the data is invalid, this constructor sets the object to a safe empty state.

`int max() const` – a private query that returns the greater of the numerator and denominator

`int min() const` – a private query that returns the lesser of the numerator and denominator

`void reduce()` – a private modifier that reduces the numerator and denominator by dividing each by their greatest common divisor

`int gcd() const` – a private query that returns the greatest common divisor of the numerator and denominator (this code has been provided)

`bool isEmpty() const` – returns true if the object is in a safe empty state; false otherwise

`void display() const` – sends the fraction to standard output in the following form only if the object holds a valid fraction and its denominator is not unity (1)

```
NUMERATOR/DENOMINATOR
```

If the object holds a valid fraction and its denominator is unity (1), your function sends

```
NUMERATOR
```

If the object is in a safe empty state, your function sends

```
no fraction stored
```

overload `operator+` as a member function – your function receives an unmodifiable reference to a `Fraction` object, which represents the right operand of the expression. The current object represents the left operand. If both of the operand are valid fractions, your function returns a copy of the result of the addition operation in reduced form. If either operand is in a safe empty state, your function returns a `Fraction` object in a safe empty state.

Using the sample implementation of the `w5_in_lab.cpp` main module shown below, test your code and make sure that it works. The expected output from your program is below the source code. The output of your program should match **exactly** the expected one.

IN-LAB MAIN MODULE

```
#include <iostream>
#include "Fraction.h"

using namespace std;
using namespace sict;

int main() {
    cout << "------------------------------" << endl;
    cout << "Fraction Class Test:" << endl;
    cout << "------------------------------" << endl;

    sict::Fraction a;
    cout << "Fraction a; // ";
    cout << "a = ";
    a.display();
    cout << endl;

    Fraction b(1, 3);
    cout << "Fraction b(1, 3); // ";
    cout << "b = ";
    b.display();
    cout << endl;

    Fraction c(-5, 15);
    cout << "Fraction c(-5, 15); //";
    cout << " c = ";
    c.display();
    cout << endl;

    Fraction d(2, 4);
    cout << "Fraction d(2, 4); //";
    cout << " d = ";
    d.display();
    cout << endl;

    Fraction e(8, 4);
    cout << "Fraction e(8, 2); //";
    cout << " e = ";
    e.display();
    cout << endl;

    cout << "a + b equals ";
    (a + b).display();
    cout << endl;

    cout << "b + d equals ";
    (b + d).display();
    cout << endl;
    return 0;
}
```

In-Lab Output

```
------------------------------
Fraction Class Test:
------------------------------
Fraction a; // a = no fraction stored
Fraction b(1, 3); // b = 1/3
Fraction c(-5, 15); // c = no fraction stored
Fraction d(2, 4); // d = 1/2
Fraction e(8, 2); // e = 2
a + b equals no fraction stored
b + d equals 5/6
```

In-Lab Submission

To test and demonstrate execution of your program use the same data as the output example above.

If not on matrix already, upload `Fraction.h`, `Fraction.cpp` and `w5_in_lab.cpp` to your matrix account. Compile and run your code and make sure everything works properly.

Then, run the following script from your account (use your professor's Seneca `userid` to replace `profname.proflastname`):

```
~profname.proflastname/submit 200_w5_lab<ENTER>
```

and follow the instructions.

IMPORTANT: Please note that a successful submission does not guarantee full credit for this workshop. If your professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.

AT-HOME (30%)

To your `Fraction` class of your in-lab solution add the following member functions and implement them in the `.cpp` file of your `Fraction` module:

overload `operator*` as a member function – your function receives an unmodifiable reference to a `Fraction` object, which represents the right operand of the expression. The current object represents the left operand. If both of the operands are valid fractions, your function returns a copy of the result of the multiplication operation in reduced form. If either operand is in a safe empty state, your function returns a `Fraction` object in a safe empty state.

overload `operator==` as a member query – your function receives an unmodifiable reference to a `Fraction` object, which represents the right operand of the expression. The current object represents the left operand. If both of the operands are valid fractions of equal value, your function returns true; otherwise false. If either operand is in a safe empty state, your function returns false.

overload `operator!=` as a member query – your function receives an unmodifiable reference to a `Fraction` object, which represents the right operand of the expression. The current object represents the left operand. If both of the operands are valid fractions of unequal value, your function returns true; otherwise false. If either operand is in a safe empty state, your function returns false.

overload `operator+=` as a member function – your function receives an unmodifiable reference to a `Fraction` object, which represents the right operand of the expression. The current object represents the left operand. If both of the operands are valid fractions, your function stores the result of the addition operation in reduced form in its instance variables and returns a reference to the current object. If either operand is in a safe empty state, your function stores a `Fraction` object in a safe empty state and returns a reference to the current object.

Code your implementation in such a way as to minimize the duplication of source code. Reuse your functions wherever possible.

Using the sample implementation of the `w5_at_home.cpp` main module shown below, test your code and make sure that it works correctly. The expected output from your program is below the source code. The output of your program should match **exactly** the expected one.

AT-HOME MAIN MODULE

```
#include <iostream>
#include "Fraction.h"

using namespace sict;
using namespace std;

int main() {
    cout << "------------------------------" << endl;
    cout << "Fraction Class Test:" << endl;
    cout << "------------------------------" << endl;

    sict::Fraction a;
    cout << "Fraction a; // ";
    cout << "a = ";
    a.display();
    cout << endl;

    Fraction b(1, 3);
    cout << "Fraction b(1, 3); // ";
    cout << "b = ";
    b.display();
    cout << endl;

    Fraction c(-5, 15);
    cout << "Fraction c(-5, 15); //";
    cout << " c = ";
    c.display();
    cout << endl;

    Fraction d(2, 4);
    cout << "Fraction d(2, 4); //";
    cout << " d = ";
    d.display();
    cout << endl;

    Fraction e(8, 4);
    cout << "Fraction e(8, 2); //";
    cout << " e = ";
    e.display();
    cout << endl;

    cout << "a + b equals ";
    (a + b).display();
    cout << endl;

    cout << "b + d equals ";
    (b + d).display();
    cout << endl;
    cout << "(b += d) equals ";
    (b += d).display();
    cout << endl;

    cout << "b equals ";
    b.display();
    cout << endl;

    cout << "(a == c) equals ";
    cout << ((a == c) ? "true" : "false");
    cout << endl;

    cout << "(a != c) equals ";
    cout << ((a != c) ? "true" : "false");
    cout << endl;

    return 0;
}
```

At-Home Output

```
------------------------------
Fraction Class Test:
------------------------------
Fraction a; // a = no fraction stored
Fraction b(1, 3); // b = 1/3
Fraction c(-5, 15); // c = no fraction stored
Fraction d(2, 4); // d = 1/2
Fraction e(8, 2); // e = 2
a + b equals no fraction stored
b + d equals 5/6
(b += d) equals 5/6
b equals 5/6
(a == c) equals false
(a != c) equals false
```

Reflection (40%)

Create a file `reflect.txt` that contains the answers to the following questions:

1. Discuss why the `operator+=` should return a reference to `Fraction`.
2. List your uses of the pointer to the current object to simplify your code.
3. Identify the temporary objects in the tester module.
4. Identify simplifications that you could make to your class without affecting the interface to the client code.
5. Explain what you have learned in this workshop.

Quiz Reflection:

Add a section to `reflect.txt` called Quiz X Reflection. Replace the X with the number of the last quiz that you received and list the numbers of all questions that you answered incorrectly.

Then for each incorrectly answered question write your mistake and the correct answer to that question. If you have missed the last quiz, then write all the questions and their answers.

At-Home Submission

To submit the *at-home* section, demonstrate execution of your program with the exact output as in the example above. Upload `reflect.txt`, `Fraction.h`, `Fraction.cpp` and `w5_at_home.cpp` to your matrix account. Compile and run your code and make sure everything works properly.

To submit, run the following script from your account and follow the instructions (use your professor's Seneca `userid` to replace `profname.proflastname`):

```
~profname.proflastname/submit 200_w5_home<ENTER>
```

Important: Please note that a successful submission does not guarantee full credit for this workshop. If the professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.