Programming Assignment #3

Learning objective : Upon completion of this program, you should gain experience with overloading basic operators for use with a C++ class. The code for this assignment should be portable -- make sure you test with g++ on program.cs.fsu.edu before you submit.

Description:

Create a class called **Mixed**. Objects of type Mixed will store and manage rational numbers in a mixed number format (integer part and a fraction part). The class, along with the required operator overloads, should be written in the files "mixed.h" and "mixed.cpp".

Programming Specifications:

1. Your class must allow for storage of rational numbers in a mixed number format. Remember that a mixed number consists of an integer part and a fraction part (like 3 1/2 -- "three and one-half"). The `Mixed` class must allow for both positive and negative mixed number values. A zero in the denominator of the fraction part constitutes an illegal number and should not be allowed. You should create appropriate member data in your class. All member data must be private.

2. There should be two constructors. One constructor should take in three parameters, representing the integer part, the numerator, and the denominator (in that order), used to initialize the object. If the mixed number is to be a negative number, the negative should be passed on the first non-zero parameter, but on no others. If the data passed in is invalid (negatives not fitting the rule, or 0 denominator), then simply set the object to represent the value 0. Examples of declarations of objects:

```
3.  Mixed m1(3, 4, 5);     // sets object to 3 4/5
4.  Mixed m2(-4, 1, 2);    // sets object to -4 1/2
5.  Mixed m3(0, -3, 5);    // sets object to -3/5 (integer part is 0).
6.  Mixed m4(-1, -2, 4);   // bad parameter combination.  Set object to 0.
```

   The other constructor should expect a single int parameter with a default value of 0 (so that it also acts as a default constructor). This constructor allows an integer to be passed in and represented as a Mixed object. This means that there is no fractional part. Example declarations:

```
Mixed m5(4);  // sets object to 4 (i.e. 4 and no fractional part).
Mixed m6;     // sets object to 0 (default)
```

   Note that this last constructor will act as a "conversion constructor", allowing automatictype conversions from type `int` to type `Mixed`.

7. The Mixed class should have public member functions **`Evaluate()`**, **`ToFraction()`**, and **`Simplify()`**. The `Evaluate()` function should return a double, the others don't return

anything. These functions have no parameters. The names must match the ones here exactly. They should do the following:

- The **Evaluate** function should return the decimal equivalent of the mixed number.
- The **Simplify** function should simplify the mixed number representation to lowest terms. This means that the fraction part should be reduced to lowest terms, and the fraction part should not be an improper fraction (i.e. disregarding any negative signs, the numerator is smaller than the denominator).
- The **ToFraction** function should convert the mixed number into fraction form. (This means that the integer part is zero, and the fraction portion may be an improper fraction).

8. Create an overload of the extraction operator `>>` for reading mixed numbers from an input stream. The input format for a `Mixed` number object will be:
9. *integer numerator/denominator*

   i.e. the integer part, a space, and the fraction part (in numerator/denominator form), where the *integer*, *numerator*, and *denominator* parts are all of type `int`. You *may* assume that this will always be the *format* that is entered (i.e. your function does *not* have to handle entry of incorrect types that would violate this format). However, this function should check the **values** that come in. In the case of an incorrect entry, just set the Mixed object to represent the number 0, as a default. An incorrect entry occurs if a denominator value of 0 is entered, or if an improper placement of a negative sign is used. Valid entry of a negative number should follow this rule -- if the integer part is non-zero, the negative sign is entered on the integer part; if the integer part is 0, the negative sign is entered on the numerator part (and therefore the negative sign should never be in the denominator). Examples:

```
Valid inputs:     2 7/3 , -5 2/7  , 4 0/7  , 0 2/5  , 0 -8/3
Invalid inputs:   2 4/0 , -2 -4/5 , 3 -6/3 , 0 2/-3
```

10. Create an overload of the insertion operator << for output of Mixed numbers. This should output the mixed number in the same format as above, with the following exceptions: If the object represents a 0, then just display a 0. Otherwise: If the integer part is 0, do not display it. If the fraction part equals 0, do not display it. For negative numbers, the minus sign is always displayed to the left.
11. `Examples:  0  ,  2  ,  -5  ,  3/4  ,  -6/7  ,  -2 4/5  ,  7 2/3`
12. Create overloads for all 6 of the comparison operators (`< , > , <= , >= , == , !=`). Each of these operations should test two objects of type Mixed and return an indication of true or false. You are testing the Mixed numbers for order and/or equality based on the usual meaning of order and equality for numbers. (These functions should **not** do comparisons by converting the Mixed numbers to decimals -- this could produce round-off errors and may not be completely accurate).

13. Create operator overloads for the 4 standard arithmetic operations (`+ , - , * , /`) , to perform addition, subtraction, multiplication, and division of two mixed numbers. Each of these operators will perform its task on two Mixed objects as operands and will return a Mixed object as a result - using the usual meaning of arithmetic operations on rational numbers. Also, each of these operators should return their result in **simplified form**. (e.g. return `3 2/3` instead of `3 10/15`, for example).

- In the division operator, if the second operand is 0, this would yield an invalid result. Since we have to return *something* from the operator, return 0 as a default (even though there is no valid answer in this case). Example:
- `Mixed m(1, 2, 3);  // value is 1 2/3`
- `Mixed z;  // value is 0`
- `Mixed r = m / z;  // r is 0 (even though this is not good math)`

14. Create overloads for the increment and decrement operators (`++` and `--`). You need to handle both the pre- and post- forms (pre-increment, post-increment, pre-decrement, post-decrement). These operators should have their usual meaning -- increment will add 1 to the Mixed value, decrement will subtract 1. Example:

```
15.    Mixed m1(1, 2, 3);          //  1 2/3
16.    Mixed m2(2, 1, 2);          //  2 1/2
17.    cout << m1++;               //  prints 1 2/3, m1 is now 2 2/3
18.    cout << ++m1;               //  prints 3 2/3, m1 is now 3 2/3
19.    cout << m2--;               //  prints 2 1/2, m2 is now 1 1/2
20.    cout << --m2;               //  prints 1/2  , m2 is now 0 1/2
```

21. **General Requirements**
    - As usual, no global variables
    - All member data of the `Mixed` class must be private
    - Use the `const` qualifier whenever appropriate
    - The only library that may be used in the class files is `iostream`
    - Since the only output involved with your class will be in the << overload (and commands to invoke it will come from some main program or other module), your output should match mine **exactly** when running test programs.

Sample main program

- I will post a sample main routine and corresponding output on the Blackboard site.

Submit the following files through the Assignment Blackboard portal:

```
mixed.h
mixed.cpp
```

Grading Criteria:

- The program compiles. If the program does not compile no further grading can be accomplished. Programs that do not compile will receive a zero.
- (25 Points) The program executes without exception and produces output. The grading of the output cannot be accomplished unless the program executes.
- (25 Points) The program produces the correct output.
- (25 Points) The program specifications are followed.

- (10 Points)The program is documented (commented) properly.
- (5 Points)Use constants when values are not to be changed
- (5 Points)Use proper indentation
- (5 Points)Use good naming standards