HW #5 (Overloading: Mixed numbers)

- 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 files "**Mixed.h**" and "**Mixed.cpp**".

- 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").

HW #5 (2)

- 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**.

- 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.

HW #5 (3)

- 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 (negative not fitting the rule, or 0 denominator), then simply set the object to represent the value 0. Examples of declarations of objects:

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

HW #5 (4)

- 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 automatic type conversions from type **int** to type **Mixed**.

HW #5 (5)

- The Mixed class should have public member functions **Evaluate()**, **ToFraction()**, and **Simplify()**.
- The **Evaluate()** function should return a double, the others do not 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.

HW #5 (6)

- 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.)

HW #5 (7)

- 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:

  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 fraction does not have to handle entry of incorrect types that would violate this format).

HW #5 (8)

- 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:

HW #5 (9)

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

- Create an overloaded 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.

  Example: 0 , 2 , -5 , 3/4 , -6/7 , -2 4/5 , 7 2/3

HW #5 (10)

- Create overloads for all 6 of the comparison operators (< , > , <= , >= , == , !=).
- Each of these operators 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.)

HW #5 (11)

- 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.

HW #5 (12)

- 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)
```

HW #5 (13)

- 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:

```
Mixed m1(1, 2, 3);          // 1 2/3
Mixed m2(2, 1, 2);          // 2 1/2
cout << m1++       // prints 1 2/3, m1 is now 2 2/3
cout << ++m1;      // prints 3 2/3, m1 is now 3 2/3
cout << m2--;      // prints 2 1/2, m2 is now 1 1/2
cout << --m2;      // prints 1/2 , m2 is now 0 1/2
```

HW #5 (14)

- As usual, no global variables.
- All member data of the Mixed class must be private.
- Use the **const** qualifier whenever appropriate.
- The only libraries that may be used in the class files are **iostream** and **iomanip**.

- The sample main program is provided. Note this is not a comprehensive set of tests. It is just a file to get you started, illustrating some sample calls.
- Two sample test runs of **main.cpp** are also provided.

```
// main.cpp
//
// Driver program to demonstrate the behavior of the Mixed class
//
// You can add more tests of your own, or write other drivers to test your
// class -- this is NOT a comprehensive set of tests.  This is just a
// sample, which should help check whether your function declarations are
// appropriate, and can test a few sample cases.

#include <iostream>
#include “Mixed.h"

using namespace std;

int main()
{
```

```
// demonstrate behavior of the two constructors and the << overload

Mixed x(3,20,6), y(-4,9,2), m1(0,-35,10), m2(-1,-2,4), m3(4), m4(-11), m5;
char answer;
cout << "Initial values: \nx = " << x << "\ny = " << y
        << "\nm1 = " << m1 << "\nm2 = " << m2 << "\nm3 = " << m3
        << "\nm4 = " << m4 << "\nm5 = " << m5 << "\n\n";

// Trying Simplify
x.Simplify();
m1.Simplify();
cout << "x simplified: " << x << " and m1 simplified " << m1 << "\n\n";

// Trying ToFraction
x.ToFraction();
y.ToFraction();
m1.ToFraction();
cout << "Values as fractions: \nx = " << x << "\ny = " << y
        << "\nm1 = " << m1 << "\n\n";
```

```
// demonstrate >> overload

cout << "Enter first number: ";
cin >> x;
cout << "Enter second number: ";
cin >> y;

cout << "You entered:\n";
cout << "  x = " << x << '\n';
cout << "  y = " << y << '\n';

// demonstrate comparison overloads
if (x < y)    cout << "(x < y) is TRUE\n";
if (x > y)    cout << "(x > y) is TRUE\n";
if (x <= y)   cout << "(x <= y) is TRUE\n";
if (x >= y)   cout << "(x >= y) is TRUE\n";
if (x == y)   cout << "(x == y) is TRUE\n";
if (x != y)   cout << "(x != y) is TRUE\n";
```

```
   // demonstrating Evaluate
   cout << "\nDecimal equivalent of " << x << " is " << x.Evaluate() << '\n';
   cout << "Decimal equivalent of " << y << " is " << y.Evaluate() << "\n\n";

   // demonstrating arithmetic overloads
   cout << "(x + y) = " << x + y << '\n';
   cout << "(x - y) = " << x - y << '\n';
   cout << "(x * y) = " << x * y << '\n';
   cout << "(x / y) = " << x / y << '\n';

   // demonstrating arithmetic that uses conversion constructor
   //  to convert the integer operand to a Mixed object
   cout << "(x + 10) = " << x + 10 << '\n';
   cout << "(x - 4) = " << x - 4 << '\n';
   cout << "(x * -13) = " << x * -13 << '\n';
   cout << "(x / 7) = " << x / 7 << '\n';

   return 0;
}
```

// run1.txt

Initial values:
x = 3 20/6
y = -4 9/2
m1 = -35/10
m2 = 0
m3 = 4
m4 = -11
m5 = 0

x simplified: 6 1/3 and m1 simplified -3 1/2

Values as fractions:
x = 19/3
y = -17/2
m1 = -7/2

Enter first number: 1 2/3
Enter second number: 2 3/4
You entered:
  x = 1 2/3
  y = 2 3/4
(x < y) is TRUE
(x <= y) is TRUE
(x != y) is TRUE

Decimal equivalent of 1 2/3 is 1.66667
Decimal equivalent of 2 3/4 is 2.75

(x + y) = 4 5/12
(x - y) = -1 1/12
(x * y) = 4 7/12
(x / y) = 20/33
(x + 10) = 11 2/3
(x - 4) = -2 1/3
(x * -13) = -21 2/3
(x / 7) = 5/21

// run2.txt

Initial values:
x = 3 20/6
y = -4 9/2
m1 = -35/10
m2 = 0
m3 = 4
m4 = -11
m5 = 0

x simplified: 6 1/3 and m1 simplified -3 1/2

Values as fractions:
x = 19/3
y = -17/2
m1 = -7/2

Enter first number: 4 3/2
Enter second number: -1 6/5
You entered:
  x = 4 3/2
  y = -1 6/5
(x > y) is TRUE
(x >= y) is TRUE
(x != y) is TRUE

Decimal equivalent of 4 3/2 is 5.5
Decimal equivalent of -1 6/5 is -2.2

(x + y) = 3 3/10
(x - y) = 7 7/10
(x * y) = -12 1/10
(x / y) = -2 1/2
(x + 10) = 15 1/2
(x - 4) = 1 1/2
(x * -13) = -71 1/2
(x / 7) = 11/14