CISS 245: Advanced Programming
Assignment 1

Objectives
The purpose of this assignment is build a simple library for a struct.

- Declare struct variable
- Access member variables of a struct variable
- Write function/operator with struct parameters
- Write function/operator with struct return value

Submission

- Hardcopy: Submit a hardcopy of your work with a coversheet.
- Softcopy: Upload your work to my Linux server, gandalf.ccis.edu.

Q1. The goal is to build a simple library for computing fractions. (To make it easy for you, I have written it in parts.)

The Given Code Base

Recall from the notes we have the `Fraction` struct. In this assignment we will be developing useful functions and operators for to support the use of this struct. We will however rename the struct as follows:

```
struct Rational
{
    int n; // numerator
    int d; // denominator
};
```

Note that C++ does not natively support fractions. (doubles are not fractions.)

You will need to write three files:

- `testRational.cpp`: This contains a program to test the `Rational` struct and it's supporting functions and operators.
- `Rational.h`: This is the header function containing the `Rational` struct and its prototypes.
- `Rational.cpp`: This file contains the definition of the prototypes in `Rational.h`.

The following are the skeleton files:

```
// Author:
// Date  :
// File  : testRational.cpp

#include <iostream>
#include "Rational.h"

int main()
{
    std::cout << "Testing Rational ...\n";

    Rational r = {1, 2};
    std::cout << "You should see 1/2 ... " << r << std::endl;
}
```

```
// Author:
// Date  :
// File  : Rational.h

#ifndef RATIONAL_H
#define RATIONAL_H

struct Rational
{
    int n; // numerator
    int d; // denominator
};

std::ostream & operator<<(std::ostream &, const Rational &)

#endif
```

```
// Author:
// Date  :
// File  : Rational.cpp

#include <iostream>
#include "Rational.h"

std::ostream & operator<<(std::ostream & cout, const Rational & r)
{
  cout << r.n << '/' << r.d;
  return cout;
}
```

You should compile and run the program to make sure that it works before continuing. Note that in Rational.cpp, the operator << is defined. Right now, the only thing you need to know is that in the code for operator<<, if you want to print `a` (say), call

```
cout << a
```

instead of

```
std::cout << a
```

in the usual C++ print statement and this is how you should print in this function. Also, do n ot modify the prototype of this “function” nor remove the last statement in the body:

```
std::ostream & operator<<(std::ostream & cout, const Rational & r)
{
  ...
  return cout;
}
```

Operators

Operators are easy to understand. They are just functions except that you call them in a different way. Let me give you an example. First run the following example:

```
#include <isotream>


struct Blah
{
  int x;
  int y;
};


double funnyOperator(const Blah & i, const Blah & j)
{
    double d;
    d = (double) i.x / i.y + (double) j.x / j.y;
    return d;
}


int main()
{
    Blah b = {2, 3};
    Blah c = {5, 9};
    double d = funnyOperator(b, c);
    std::cout << d << std::endl;
    return 0;
}
```

(The actual body of the function is not important. Focus on how the function is called.) There are no surprises here. Now try this:

```
#include <iostream>


struct Blah
{
  int x;
  int y;
};


double operator+(const Blah & i, const Blah & j)
{
    double d;
    d = (double) i.x / i.y + (double) j.x / j.y;
    return d;
}


int main()
{
    Blah b = {2, 3};
    Blah c = {5, 9};
    double d = operator+(b, c);
    std::cout << d << std::endl;
```

```
    return 0;
}
```

The name of the function, `funnyOperator`, is changed to `operator+`.

And finally run this program:

```
#include <iostream>


struct Blah
{
  int x;
  int y;
};


double operator+(const Blah & i, const Blah & j)
{
    double d;
    d = (double) i.x / i.y + (double) j.x / j.y;
    return d;
}


int main()
{
    Blah b = {2, 3};
    Blah c = {5, 9};
    double d = b + c;
    std::cout << d << std::endl;
    return 0;
}
```

As you can see in the above program

```
b + c
```

is really a call to the "function" operator+:

```
operator+(b, c)
```

except that the "function" operator+, in this context, is really called an **operator**. Once again the following are actually the same:

```
b + c                operator+(b, c)
```

You can define all kinds of operators in C++, including

\>\> \<\< + – * /

etc.

A simple `operator<<` is already defined for you.

Q1(a). operator<<

Modify operator<< so that:

- Negative signs are printed correctly, i.e. either none or once and never twice. For instance if the Rational variable x has x.n = -2 and x.d = -4,

  ```
  Rational x = {-2, -4};
  std::cout << x << std::endl;
  ```

  will display

  ```
  2/4
  ```

  instead of

  ```
  -2/-4
  ```

- There should not be any negative sign for the print out of the denominator. For instance if the Rational variable x has x.n = 5 and x.d = -2,

  ```
  Rational x = {5, -2};
  std::cout << x << std::endl;
  ```

  then will display

  ```
  -5/2
  ```

  instead of

  ```
  5/-2
  ```

- If the denominator is 1 or -1, only an integer value is printed. For instance if the Rational variable x has x.n = -3 and x.d = -1,

  ```
  Rational x = {-3, -1};
  std::cout << x << std::endl;
  ```

  will display

  ```
  3
  ```

  instead of

  ```
  -3/-1
  ```

- If the numerator is 0, then `0` is printed. In other words

  ```
  Rational x = {0, -5000};
  std::cout << x << std::endl;
  ```

  will display

  ```
  0
  ```

- If the denominator is 0, then `UNDEFINED` is printed. In other words

  ```
  Rational x = {123, 0};
  std::cout << x << std::endl;
  ```

  will display

  ```
  UNDEFINED
  ```

Add some test cases to testRational.cpp:

```
// Author:
// Date  :
// File  : testRational.cpp
```

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

int main()
{
    std::cout << "Testing Rational ...\n";

    Rational r = {1, 2};
    std::cout << "You should see 1/2 ... " << r << std::endl;

    r.n = -1;
    r.d = -2;
    std::cout << "You should see 1/2 ... " << r << std::endl;

    r.n = 3;
    r.d = 1;
    std::cout << "You should see 3 ... " << r << std::endl;

    r.n = 3;
    r.d = -1;
    std::cout << "You should see -3 ... " << r << std::endl;
}
```

(I strongly strongly advice you to add as many test cases as you can and test your code as thoroughly as possible.)

Q1(b). operator+

Define operator+ so that when operator+ is called with two Rational variables, the return value is a Rational value that models the addition of two rational numbers in the “real” world. Mathematically this is how you add two rational numbers:

```
a   c   ad + bd
- + - = -------
b   d     bd
```

Note that operator+ accepts two Rational values and returns a Rational value therefore you have to add this to the Rational header file:

```
// Author:
// Date  :
// File  : Rational.h

#ifndef RATIONAL_H
#define RATIONAL_H

struct Rational
{
    int n; // numerator
    int d; // denominator
};

std::ostream & operator<<(std::ostream &, const Rational &)
Rational operator+(const Rational &, const Rational &);

#endif
```

Add the following tests to your main():

```
// Author:
// Date  :
// File  : testRational.cpp

#include <iostream>
#include "Rational.h"

int main()
{
    std::cout << "Testing Rational ...\n";

    Rational r = {1, 2};
    std::cout << "You should see 1/2 ... " << r << std::endl;

    r.n = -1;
    r.d = -2;
    std::cout << "You should see 1/2 ... " << r << std::endl;

    r.n = 3;
    r.d = 1;
    std::cout << "You should see 3 ... " << r << std::endl;

    r.n = 3;
    r.d = -1;
```

```
    std::cout << "You should see -3 ... " << r << std::endl;

    r.n = 3;
    r.d = -1;
    std::cout << "You should see -3 ... " << r << std::endl;

    Rational a = {1, 2};
    Rational b = {1, 4};
    std::cout << "You should see 6/8 ... " << a + b << std::endl;

    Rational c = {1, 2};
    Rational d = {1, 4};
    Rational e = {-1, -2};
    std::cout << "You should see 6/8 ... " << a + b << std::endl;
    std::cout << "You should see 6/8 ... " << b + a << std::endl;
    std::cout << "You should see 6/8 ... " << c + b << std::endl;
}
```

HINT: SPOILERS AHEAD … READ ONLY WHEN YOU REALLY NEED IT ...

Mathematically this is how we add fractions:

```
          a   c   ad + bd
          - + - = -------
          b   d     bd
```

If x and y are the parameters in the body of operator+, then conceptually in terms of the members of x and y we have:

```
                  x.n   y.n   (x.n) * (y.d) + (x.d) * (y.n)
          x + y = --- + --- = -----------------------------
                  x.d   y.d           (x.d) * (y.d)
```

The operator looks like this:

```
Rational operator+(const Rational & x, const Rational & y)
{
    Rational sum;
    return sum;
}
```

Of course you need to set the numerator and denominator of the value to return:

```
Rational operator+(const Rational & x, const Rational & y)
{
    Rational sum = {???, ???};
    return sum;
}
```

Q1(c). operator−

Once you're understood operator+, this part is easy.

Define operator- so that when operator- is called with two Rational variables, the return value is a Rational value that models the difference of rational numbers in the “real” world. Mathematically this is how you add two rational numbers:

```
a   c   ad - bd
- - - = -------
b   d     bd
```

Add more test cases to your main().

Q1(d). operator*

Define operator* so that when operator* is called with two Rational variables, the return value is a Rational value that models the product of rational numbers in the “real” world. Mathematically this is how you multiply rational numbers:

```
a   c   a * c
- * - = -----
b   d   b * d
```

Add more test cases to your main().

Q1(e). operator/

Define operator* so that when operator* is called with two Rational variables, the return value is a Rational value that models the product of rational numbers in the “real” world. Mathematically this is how you multiply rational numbers:

```
a   c   a * d
- / - = -----
b   d   b * c
```

Add more test cases to your main().

Q1(f). operator==

Define operator== so that when operator== is called with two Rational variables, the return value is true exactly then the Rational values have the same mathematical value. Mathematically two rational numbers

```
a       c
-  and  -
b       d
```

are the same, i.e.

```
a   c
- = -
b   d
```

if

```
a * d = b * c                                        (A)
```

Note that this is ***not*** the same as

```
a = c and b = d                                      (B)
```

which is ***incorrect***. For instance the fractions 1/2 and 2/4 are the same which is correctly detected by condition (A) but not (B).

Add more test cases to your main().

Q1(g). operator!=

Now add operator!= to your code. Note that if x and y are Rational values, then

```
x != y
```

is the same as

```
operator!=(x, y)
```

The prototype of `operator!=()` is

```
bool operator!=(const Rational &, const Rational &);
```

It's very important to note the following: Instead of implementing operattor!= from scratch, your operator!= MUST use operator==.

Add more test cases to your main().

HINT: operator!= is just the “opposite” of operator==.

Q1(h). reduce() function

Note that all the above functions are operators. For this part, you need to write a function. This function reduces the Rational value to its lowest terms. Mathematically, the fraction

```
18
--
60
```

is not a reduced (or simplified) fraction. That's because 2 is a factor of 18 and 60:

```
18   2 * 9
-- = ------
60   2 * 30
```

If 2 is removed from the numerator and denominator, we get

```
18   2 * 9     9
-- = ------ = --
60   2 * 30   30
```

This is still not reduced since 3 is a factor of 9 and 30. Removing 3 we get

```
 9    3 * 3     3
--  = ------ = --
30    3 * 10   10
```

This fraction, 3/10, is now reduced: you cannot find any integer factor of both 3 and 10 other than 1 and -1.

For this part you need to implement

```
void reduce(Rational &);
```

that reduces the reference parameter so that the fraction is in its reduced form. For instance

```
Rational r = {18, 60}
reduce(r);
// at this point r.n = 3 and r.d = 10
```

It's not enough just to remove a single or two common divisor; you must remove ***all*** common divisors which are greater than 1. There are a couple of other points:

- The reduce() function must also ensure that the denominator of the parameter is positive:

```
Rational r = {1, -2}
reduce(r);
// at this point r.n = -1 and r.d = 2

Rational s = {-1, -2}
reduce(s);
// at this point s.n = 1 and s.d = 2
```

- If the numerator is 0, the denominator is set to 0

```
Rational r = {0, -1500}
reduce(r);
// at this point r.n = 0 and r.d = 1
```

.

- If the denominator is 0 (the case where the fraction is not defined), the numerator is set to 1:

```
Rational r = {-42, 0}
reduce(r);
// at this point r.n = 1 and r.d = 0.
```

Add more test cases to your main().

HINT: If d divides a and b, then

```
a   a / d
- = -----
b   b / d
```

For instance 2 divides 18 and 60. Therefore

```
18   18 / 2    9
-- = ------ = --
60   60 / 2   30
```

Your code should find the greatest common divisors g of the numerator and denominator of a fraction, and then divide the numerator and denominator of this fraction by g. For this assignment you need not worry about efficiency.