Problem 3: Rational Number Library (rational)

Now that we know about the rounding issues of floating-point arithmetic, we'd like to implement a library that will allow for exact computation with rational numbers. This library should be easy and intuitive to use. You want to be able to write code as straight-forward as you can see in this sample:

```
auto a = Rational{ 1, 3 }; // the number 1/3
auto b = Rational{ -6, 7 }; // the number -6/7
std::cout << "a * b = " << a << " * " << b << "       " << a * b << "     ;
std::cout << "a / ( a + b / a ) = " << a / ( a + b / a ) << "    ;
```

which produces the following as output.

```
a * b = 1/3 * -6/7 = -2/7
a / ( a + b / a ) = -7/47
```

Of course, we'll be able to do much more impressive things. For example, here is a snippet of code that computes the golden ratio φ to over 15 decimal digits of accuracy using a very readable implementation of its continued fraction expansion.

$$\varphi = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \cdots}}} \approx 1.618033\ldots$$

```
const int ITER = 40;
auto phi = Rational{ 1 }; // set equal to 1 to start off with
for( int i = 0; i < ITER; ++i ) {
  phi = 1 / ( 1 + phi );
}
std::cout << std::setprecision( 15 ); // look up <iomanip>
std::cout << "phi = " << ( 1 + phi ).to_double() << "    ;
```

This prints out `phi = 1.61803398874989`, which happens to be 15 completely accurate digits.

You can find the zipped project at provided in the file rational.zip as a basis for your program. As in the other project problem, the folder contains an `include/` directory, our standard project Makefile, and a `src/` directory. All of our header (.h) files will be found in `include/` while all of our source (.cpp) files will be in `src/`. The C++ files you should find are `rational.h`, `gcd.h`, `rational.cpp`, `gcd.cpp`, and `test.cpp`; you will be modifying `rational.h` and `rational.cpp`. Like in the previous project problem, you can modify the Makefile or `test.cpp` to run your own tests, but your changes there will be overwritten when you upload to the grader.

We are provided a header file describing the interface for the **Rational** class.

```
#ifndef _6S096_RATIONAL_H
#define _6S096_RATIONAL_H

#include <cstdint>
#include <iosfwd>
#include <stdexcept>

class Rational {
  intmax_t _num, _den;
public:
  enum sign_type { POSITIVE, NEGATIVE };

  Rational() : _num{0}, _den{1} {}
  Rational( intmax_t numer ) : _num{numer}, _den{1} {}
  Rational( intmax_t numer, intmax_t denom ) : _num{numer}, _den{denom} {
    normalize(); // reduces the Rational to lowest terms
  }

  inline intmax_t num() const { return _num; }
  inline intmax_t den() const { return _den; }

  // you'll implement all these in rational.cpp
  void normalize();
  float to_float() const;
  double to_double() const;
  sign_type sign() const;
  Rational inverse() const;
};

// An example multiplication operator
inline Rational operator*( const Rational &a, const Rational &b ) {
  return Rational{ a.num() * b.num(), a.den() * b.den() };
}

// Functions you'll be implementing:
std::ostream& operator<<( std::ostream& os, const Rational &ratio );
// ... and so on
```

Notice the include guards at the top, protecting us from including the file multiple times. An explanation of some of the constructs we use is in order. You'll notice we use the `intmax_t` type from the `<cstdint>` library. This type represents the maximum width integer type for our particular architecture (basically, we want to use the largest integers available). On my 64-bit computer, this means that we'lll

find `sizeof(Rational)` to be 16 since it comprises two 64-bit (8-byte) integers.

Look up an enum. We use this to define a type `sign_type` in the scope of our class. Outside the class, we can refer to the type as `Rational::sign_type` and its two values as `Rational::POSITIVE` and `Rational::NEGATIVE`.

Contained in the header file is also a class that extends `std::domain_error` to create a special type of exception for our **Rational** class. Look up the meaning of explicit; why do we use it in this particular constructor definition?

```
#include <stdexcept>
// ...near the bottom of the file
class bad_rational : public std::domain_error {
public:
  explicit bad_rational() : std::domain_error( "Bad rational: zero denom." ) {}
};
```

Read through the code carefully and make sure you understand it. If you have any questions about the provided code or don't know why something is structured the way it is, please ask about it on Piazza.

This is the list of functions you should fill out. In the header file, you have a number of `inline` functions to complete, as shown here.

```
inline Rational operator+( const Rational &a, const Rational &b ) {/* ... */}
inline Rational operator-( const Rational &a, const Rational &b ) {/* ... */}
inline Rational operator/( const Rational &a, const Rational &b ) {/* ... */}
inline bool operator<( const Rational &lhs, const Rational &rhs ) {/* ... */}
inline bool operator==( const Rational &lhs, const Rational &rhs ) {/* ... */}
```

There are also a number of functions declared in the header file which you should write an implementation for in the source file `rational.cpp`. Those declarations are:

```
class Rational {
  // ...snipped
public:
  // ...snipped
  void normalize();
  float to_float() const;
  double to_double() const;
  sign_type sign() const;
};
std::ostream& operator<<( std::ostream& os, const Rational &ratio );
```

Look in the existing `rational.cpp` file for extensive descriptions of the required functionality.

In `gcd.h`, you can find the declaration for two functions which you may find useful: fast computation of the greatest common divisor and least common multiple of two integers. Feel free to include this header and use these functions in your code as needed.

```
#ifndef _6S096_GCD_H
#define _6S096_GCD_H

#include <cstdint>

intmax_t gcd( intmax_t a, intmax_t b );
intmax_t lcm( intmax_t a, intmax_t b );

#endif // _6S096_GCD_H
```

Input Format

Not applicable; your library will be compiled into a testing suite, your implemented functions will be called by the program, and the behavior checked for correctness. For example, here is a potential test:

```
#include "rational.h"
#include <cassert>

void test_addition() {
  // Let's try adding 1/3 + 2/15 = 7/15.
  auto sum = Rational{ 1, 3 } + Rational{ 2, 15 }
  assert( sum.num() == 7 );
  assert( sum.den() == 15 );
}
```

You are strongly encouraged to write your own tests in `test.cpp` so that you can try out your implementation code before submitting it to the online grader.

Output Format

Not applicable.

6.S096 Effective Programming in C and C++

IAP 2014