**Computer Science 1107**

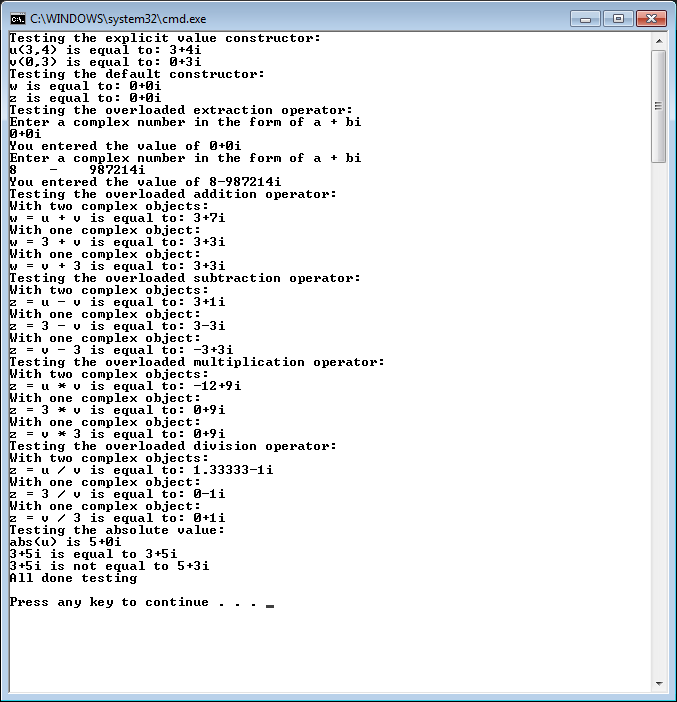
**Lab Experience Seven**

A **complex number** is a [number](https://en.wikipedia.org/wiki/Number) that can be expressed in the form *a* + *bi*, where *a* and *b* are [*real numbers*](https://en.wikipedia.org/wiki/Real_number) and *i* is the imaginary unit, that satisfies the equation *i*2 = −1. In this expression, *a* is called the *real part* and *b* is called the *imaginary part* of the complex number.

The following are operations that can be performed on Complex Numbers:

|  |  |
| --- | --- |
| Operation | Formula |
| Addition | (a+bi) + (c+di) = (a+c) + (b+d)i.\ |
| Subtraction | (a+bi) - (c+di) = (a-c) + (b-d)i.\ |
| Multiplication | (a+bi) (c+di) = (ac-bd) + (bc+ad)i.\ |
| Division | \,\frac{a + bi}{c + di} = \left({ac + bd \over c^2 + d^2}\right) + \left( {bc - ad \over c^2 + d^2} \right)i. |
| Equality | (a + bi) == (c + di) iff a == c and b == d. |
| Absolute Value where z = x + yi and r is a real number. | \textstyle r=|z|=\sqrt{x^2+y^2}.\, |

Write a class called Complex that implements the above methods. You should also overload the insertion and the extraction operator. The driver routine is provided for you.



/\* complex.h

\* Defines a class Complex.

\* Johnathan Lee CSCI 1107

\* Due 03/06/18

\*/

#ifndef COMPLEX\_H

#define COMPLEX\_H

#include <cmath>

#include <iostream>

using namespace std;

/// \class Complex

/// \brief A class for handling Complex numbers.

class Complex {

public:

/// \brief Default/explicit combo. Initializes to 0+0i by default

/// \param inReal number to initialize this->real to.

/// \param inImagi number to initialize this->imagi to.

**Complex**(double inReal = 0,

double inImagi = 0); // Construct from doubles (Default)

/// \brief Displays the Complex number to the out stream.

/// \param out The stream to display to.

/// \post This number is printed to the stream in 1 of the 2 formats:

/// <real>+<imagi>i, if imagi is pos, and

/// <real>-<imagi>i, if imagi is neg.

void **display**(ostream& out) const;

/// \brief Reads a complex number from an input stream

/// \post \*this has been replaced by whatever is next in the stream.

void **read**(istream& in);

/// \brief Adds \*this to another Complex.

/// \param rhs The right hand side

/// \returns \*this + rhs, following rules for complex arithmetic.

Complex **add**(const Complex& rhs) const;

/// \brief Subtracts rhs from \*this.

/// \param rhs The right hand side

/// \returns \*this - rhs.

Complex **subtract**(const Complex& rhs) const;

/// \brief Multiplies \*this by another complex

/// \param rhs The right hand side

/// \returns \*this \* rhs, following rules for complex arithmetic.

Complex **mult**(const Complex& rhs) const;

/// \brief Divides \*this by another Complex.

/// \param rhs The right hand side

/// \returns \*this / rhs, following rules for complex arithmetic.

Complex div(const Complex& rhs) const;

/// \brief Returns whether this is equal to another Complex.

/// \param rhs The Complex to compare ourselves with.

/// \returns this->real == rhs.real && this->imagi == rhs.imagi.

bool **isEqual**(const Complex& rhs) const;

/// \brief Getter for this->real

double **getReal**() const;

/// \brief Getter for this->imagi

double **getImagi**() const;

/// \brief Setter for this->real

/// \note Sets directly, no special rules.

void **setReal**(double newReal);

/// \brief Setter for this->imagi

/// \note Sets directly, no special rules.

void **setImagi**(double newImagi);

private:

double real, ///> The real part of a+bi (a)

imagi; ///> The imaginary part of a+bi (b)

};

// OVERLOADS OF FUNCTIONS FOR COMPLEX

/// \brief Overload of absolute value function for Complex numbers.

/// \param com The number to take an absolute value of.

/// \returns The absolute value of com, following rules of complex arithmetic.

Complex abs(const Complex& com);

// COMPLEX OPERATORS

// Note: These all directly call the equivalent functions in lhs, passing in rhs

// as a param.

Complex *operator*+(const Complex& lhs, const Complex& rhs);

Complex *operator*-(const Complex& lhs, const Complex& rhs);

Complex *operator*\*(const Complex& lhs, const Complex& rhs);

Complex *operator*/(const Complex& lhs, const Complex& rhs);

bool *operator*==(const Complex& lhs, const Complex& rhs);

/// \brief Reads in a complex number from in to com

/// \note Simply calls com.read(in)

istream& *operator*>>(istream& in, Complex& com);

/// \brief Displays a complex number com to a stream out.

/// \note Simply calls com.display(out)

ostream& *operator*<<(ostream& out, const Complex& com);

#endif

// complex.cpp

// Implemenation file for complex.h

#include "complex.h"

// DEFAULT AND EXPLICIT CONSTRUCTOR

Complex::Complex(double inReal, double inImagi) {

real = inReal;

imagi = inImagi;

}

// IO FUNCS

void Complex::display(ostream& out) const {

out << real << (imagi >= 0 ? "+" : "") << imagi << "i";

}

void Complex::read(istream& in) {

char dummy;

// a+bi, put a/b into real/imagi

in >> real >> dummy >> imagi;

// Assuming no spaces, a simple in>>real>>imagi>>dummy would work,

// but if they enter as a - bi, it would fail and give a + 0i, so check sign.

if (dummy == '-')

imagi \*= -1;

// Consume i

in >> dummy;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// NOTE: IN THE COMMENTS FOR EACH OF THESE, A and C REFER TO THE LHS AND RHS

// REAL PARTS, WHILE B AND D REFER TO THE LHS AND RHS IMAGI PARTS.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// OPERATOR FUNCS

Complex Complex::add(const Complex& rhs) const {

// (a+c)+(b+d)i

return Complex(real + rhs.real, imagi + rhs.imagi);

}

Complex Complex::subtract(const Complex& rhs) const {

return add(rhs.mult(-1)); // Addition is subtraction of opposite.

}

Complex Complex::mult(const Complex& rhs) const {

// (ac - bd) + (bc + ad)i

return Complex(real \* rhs.real - imagi \* rhs.imagi,

imagi \* rhs.real + real \* rhs.imagi);

}

Complex Complex::div(const Complex& rhs) const {

double newReal, newImagi, // You know it's getting serious when we split

// this into variables first.

denom =

(pow(rhs.real, 2) + pow(rhs.imagi, 2)); // Since they're the same.

// (ac+bd)/(c^2+d^2)

newReal = (real \* rhs.real + imagi \* rhs.imagi) / denom;

newImagi = (imagi \* rhs.real - real \* rhs.imagi) / denom;

return Complex(newReal, newImagi);

}

bool Complex::isEqual(const Complex& rhs) const {

return (real == rhs.real) && (imagi == rhs.imagi);

}

// GETTERS AND SETTERS

double Complex::getReal() const {

return real;

}

double Complex::getImagi() const {

return imagi;

}

void Complex::setReal(double newReal) {

real = newReal;

}

void Complex::setImagi(double newImagi) {

imagi = newImagi;

}

// NON MEMBER FUNCS

// No idea why this would be a non member func....

Complex abs(const Complex& com) {

return sqrt(pow(com.getReal(), 2) + pow(com.getImagi(), 2));

}

// OPERATORS

Complex *operator*+(const Complex& lhs, const Complex& rhs) {

return lhs.add(rhs);

}

Complex *operator*-(const Complex& lhs, const Complex& rhs) {

return lhs.subtract(rhs);

}

Complex *operator*\*(const Complex& lhs, const Complex& rhs) {

return lhs.mult(rhs);

}

Complex *operator*/(const Complex& lhs, const Complex& rhs) {

return lhs.div(rhs);

}

bool *operator*==(const Complex& lhs, const Complex& rhs) {

return lhs.isEqual(rhs);

}

istream& *operator*>>(istream& in, Complex& com) {

com.read(*in*);

return in;

}

ostream& *operator*<<(ostream& out, const Complex& com) {

com.display(*out*);

return out;

}

#include "complex.h"

#include <iostream>

using namespace std;

int **main**(){

Complex u(3, 4), v(0,3);

cout << "Testing the explicit value constructor: " << endl;

cout << "u(3,4) is equal to: " << u << endl

<< "v(0,3) is equal to: " << v << endl;

Complex w, z;

cout << "Testing the default constructor: " << endl;

cout << "w is equal to: " << w << endl

<< "z is equal to: " << z << endl;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

cout << "Testing the overloaded extraction operator: " << endl;

cout << "Enter a complex number in the form of a + bi" << endl;

cin >> w;

cout << "You entered the value of " << w << endl;

cout << "Enter a complex number in the form of a + bi" << endl;

cin >> z;

cout << "You entered the value of " << z << endl;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

cout << "Testing the overloaded addition operator: " << endl;

cout << "With two complex objects: " << endl;

w = u + v;

cout << "w = u + v is equal to: " << w << endl;

cout << "With one complex object: " << endl;

w = 3 + v;

cout << "w = 3 + v is equal to: " << w << endl;

cout << "With one complex object: " << endl;

w = v + 3;

cout << "w = v + 3 is equal to: " << w << endl;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

cout << "Testing the overloaded subtraction operator: " << endl;

cout << "With two complex objects: " << endl;

z = u - v;

cout << "z = u - v is equal to: " << z << endl;

cout << "With one complex object: " << endl;

z = 3 - v;

cout << "z = 3 - v is equal to: " << z << endl;

cout << "With one complex object: " << endl;

z = v - 3;

// NOTE: Changed from v+3 to v-3 in string to match above op.

cout << "z = v - 3 is equal to: " << z << endl;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

cout << "Testing the overloaded multiplication operator: " << endl;

cout << "With two complex objects: " << endl;

z = u \* v;

cout << "z = u \* v is equal to: " << z << endl;

cout << "With one complex object: " << endl;

z = 3 \* v;

cout << "z = 3 \* v is equal to: " << z << endl;

cout << "With one complex object: " << endl;

z = v \* 3;

cout << "z = v \* 3 is equal to: " << z << endl;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

cout << "Testing the overloaded division operator: " << endl;

cout << "With two complex objects: " << endl;

z = u / v;

cout << "z = u / v is equal to: " << z << endl;

cout << "With one complex object: " << endl;

z = 3 / v;

cout << "z = 3 / v is equal to: " << z << endl;

cout << "With one complex object: " << endl;

z = v / 3;

cout << "z = v / 3 is equal to: " << z << endl;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

cout << "Testing the absolute value: " << endl;

cout << "abs(u) is " << abs(u) << endl;

Complex c1(3, 5), c2(3, 5), c3(5, 3);

if (c1 == c2)

cout << c1 << " is equal to " << c2 << endl;

else

cout << c1 << " is not equal to " << c2 << endl;

if (c2 == c3)

cout << c2 << " is equal to " << c3 << endl;

else

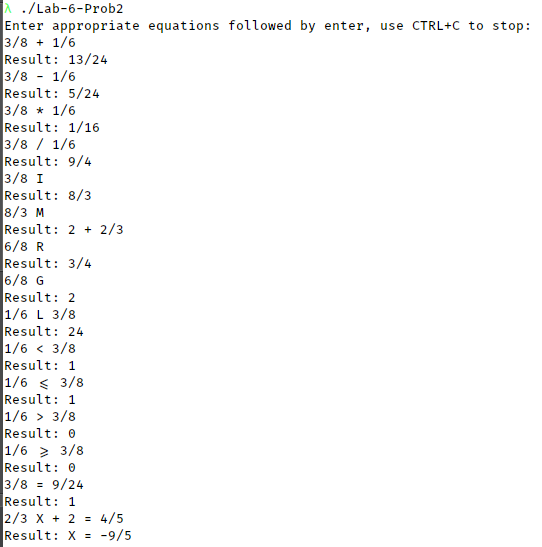
cout << c2 << " is not equal to " << c3 << endl;

cout << "All done testing\n\n";

return 0;

}

Lab 6, Problem 2:



/\* Lab6-Prob2.cpp

\* Test driver for RationalNumber class.

\* NOTE: Includes an auto tester. Simply define AUTO\_TESTS

\*

\* Johnathan Lee CSCI 1107

\* Due: 02/26/18

\*/

#include <cassert>

#include <iostream>

#include <sstream>

using namespace std;

#include "RationalNumber.h"

// Because typing in test inputs over and over during development got boring.

#define AUTO\_TESTS

// Tester for RationalNumber

int **main**() {

RationalNumber num;

cout << "Enter appropriate equations followed by enter, use CTRL+C to "

"stop: \n";

#ifdef AUTO\_TESTS

// All of the official inputs for easy automated testing.

std::string inputs[] = {"3/8 + 1/6\n", "3/8 - 1/6\n", "3/8 \* 1/6\n",

"3/8 / 1/6\n", "3/8 I\n", "8/3 M\n",

"6/8 R\n", "6/8 G\n", "1/6 L 3/8\n",

"1/6 < 3/8\n", "1/6 <= 3/8\n", "1/6 > 3/8\n",

"1/6 >= 3/8\n", "3/8 = 9/24\n", "2/3 X + 2 = 4/5\n"};

std::string expectedOutputs[] = {

"13/24", "5/24", "1/16", "9/4", "8/3", "2 + 2/3", "3/4", "2",

"24", "1", "1", "0", "0", "1", "X = -9/5"};

for (int i = 0; i < 15; i++) {

// You're right, this class is pretty useful.

stringstream testIn, testOut;

testIn << inputs[i];

num.parse(testIn, testOut);

cout << inputs[i] << "Result: ";

assert(expectedOutputs[i] == testOut.str());

cout << testOut.str();

cout << "\n";

}

#else

while (true) {

num.parse(cin, cout);

cout << "\n";

}

#endif

return 0;

}

/\* RationalNumber.h

\* Defines a class RationalNumber and its functions/operators.

\* Johnathan Lee CSCI 1107

\* Due 03/06/18

\*/

#ifndef RATIONALNUMBER\_H

#define RATIONALNUMBER\_H

#include <cmath>

#include <iostream>

#include <sstream>

using namespace std;

/// \class RationalNumber

/// \brief A class for handling rational numbers of the form a/b, where a and b

/// are integers and b is non-zero.

class RationalNumber {

public:

/// \brief Explicit and default constructor for RationalNumber

/// \post \*this has been constructed with either pA and pB for numerator and

/// denominator

**RationalNumber**(int pA = 0, int pB = 1);

/// \brief Reads in a RationalNumber from in

/// \param in Stream to read from

/// \post \*this is set to the input.

/// \note Input is in the form of either a/b OR simply a, which becomes a/1.

void **read**(istream& in); // For a single rational number

/// \brief Parses an equation from in, sets this to result, and prints

/// result.

///

/// \param in Stream to read from.

/// \param out Stream to send result to.

/// \post \*this has been set to the result of the operation, \*this has been

/// sent to out.

/// \note Supported operators:

/// Binary: + - \* / < <= > >= = L

/// Unary: I M R G

/// Expression: a/bX + c/d = e/f (Solves for X)

void **parse**(istream& in,

ostream& out); // For a full equation with operations

/// \brief Outputs \*this to out.

/// \param out Stream to display to.

/// \post \*this Has been written to out in one of the following forms:

/// a/b, if b != 1

/// a, if b == 1

void **display**(ostream& out) const; // Write to output.

// Unary operations

/// \brief Inverts \*this from a/b to b/a and returns it.

RationalNumber **invert**() const;

/// \brief Finds the greatest common factor of this->num and this->denom and

/// returns it.

int **greatestCommonFactor**() const;

/// \brief Reduces \*this. (i.e 2/4->1/2, 4/2->2/1, etc)

/// \note Worth noting that this internally calls

/// this->greatestCommonFactor().

RationalNumber **reduced**() const;

/// \brief Returns a string which represents this as a mixed form, in the

/// form m + a/b. (i.e 5/2 -> 2 + 1/2)

string **mixedForm**() const;

// Binary operations

/// \brief Adds \*this to rhs and returns the result.

/// \param rhs The right-hand-side of the equation lhs + rhs.

RationalNumber **add**(const RationalNumber& rhs) const;

/// \brief Subtracts rhs from \*this and returns the result.

/// \param rhs The right-hand-side of the equation lhs - rhs.

/// \note Internally calls add for addition of the opposite.

RationalNumber **sub**(const RationalNumber& rhs) const;

/// \brief Multiplies \*this by rhs and returns the result.

/// \param rhs The right hand side of the equation lhs \* rhs.

RationalNumber **mult**(const RationalNumber& rhs) const;

/// \brief Divides \*this by rhs and returns the result.

/// \param rhs The right hand side of the equation lhs / rhs.

/// \note Internally calls mult on rhs.invert()

RationalNumber div(const RationalNumber& rhs) const;

/// \brief Finds the lowest common denominator of \*this and rhs

/// \param rhs The other number to find the LCD for.

/// \returns The lowest common denominator of lhs and rhs.

int **lowestCommonDenominator**(const RationalNumber& rhs) const;

// Relational Operations

/// \brief Less than comparison

/// \returns a/b < c/d, where a/b and c/d are \*this and rhs in reduced form

/// with a common denominator.

bool **isLessThan**(const RationalNumber& rhs) const;

/// \brief Less than or equal to comparison

/// \returns a/b <= c/d, where a/b and c/d are \*this and rhs in reduced form

/// with a common denominator.

bool **isLessThanEqual**(const RationalNumber& rhs) const;

/// \brief Greater than comparison

/// \returns a/b > c/d, where a/b and c/d are \*this and rhs in reduced form

/// with a common denominator.

bool **isGreaterThan**(const RationalNumber& rhs) const;

/// \brief Greater than or equal to comparison

/// \returns a/b >= c/d, where a/b and c/d are \*this and rhs in reduced form

/// with a common denominator.

bool **isGreaterThanEqual**(const RationalNumber& rhs) const;

/// \brief Equivalence comparison

/// \returns Whether the reduced form of \*this is equal to the reduced form

/// of rhs. Note: Does a direct comparison of reducedThis and reducedRhs

/// member variables.

bool **isEqual**(const RationalNumber& rhs) const;

/// \brief Getter for this->num

int **getNum**() const;

/// \brief Setter for this->num

/// \param value The new value for a

/// \post this->num = value

void **setNum**(int value);

/// \brief Getter for this->denom

int **getDenom**() const;

/// \brief Setter for this->denom

/// \param value The new value for b

/// \post this->denom = value. If value == 0, then b is set to 1. If b < 0,

/// then a has its sign flipped.

void **setDenom**(int value);

/// \brief Convenience setter for both numerator and denominator

/// \param valueA The new value for this->num

/// \param valueB The new value for this->denom

/// \post this->num = valueA and this->denom = valueB.

void **setAB**(int valueA, int valueB);

private:

int num, ///< The numerator of the fraction

denom; ///< The denominator of the fraction.

};

//---------->RationalNumber Operators

// IO

/// \brief Reads into a RationalNumber

/// \param in The stream to read from

/// \param num The number to store in.

/// \post num.read(in) has been called.

istream& *operator*>>(istream& in, RationalNumber& num);

/// \brief Writes out a RationalNumber

/// \param out The stream to write to.

/// \param num The number to read from.

/// \post num.display(out) has been called.

ostream& *operator*<<(ostream& out, const RationalNumber& num);

// Arithmetic

// NOTE: These are all nothing but wrappers for RationalNumber methods.

/// \brief Calls lhs.add(rhs)

RationalNumber *operator*+(const RationalNumber& lhs, const RationalNumber& rhs);

/// \brief Calls lhs.sub(rhs)

RationalNumber *operator*-(const RationalNumber& lhs, const RationalNumber& rhs);

/// \brief Calls lhs.mult(rhs)

RationalNumber *operator*\*(const RationalNumber& lhs, const RationalNumber& rhs);

/// \brief Calls lhs.div(rhs)

RationalNumber *operator*/(const RationalNumber& lhs, const RationalNumber& rhs);

// Relational

/// \brief Calls lhs.isLessThan(rhs)

bool *operator*<(const RationalNumber& lhs, const RationalNumber& rhs);

/// \brief Calls lhs.isLessThanEqual(rhs)

bool *operator*<=(const RationalNumber& lhs, const RationalNumber& rhs);

/// \brief Calls lhs.isGreaterThan(rhs)

bool *operator*>(const RationalNumber& lhs, const RationalNumber& rhs);

/// \brief Calls lhs.isGreaterThanEqual(rhs)

bool *operator*>=(const RationalNumber& lhs, const RationalNumber& rhs);

/// \brief Calls lhs.isEqual(rhs)

bool *operator*==(const RationalNumber& lhs, const RationalNumber& rhs);

#endif

// RationalNumber.cpp

// Implemenation file for RationalNumber.h

#include "RationalNumber.h"

#include <cassert>

RationalNumber::RationalNumber(int pA, int pB) {

// Initialize to 0=0/1

num = pA;

denom = pB;

}

// I/O

void RationalNumber::read(istream& in) {

in >> num; // get a

// So we can input single numbers (like the eq solver's input)

if (in.peek() == '/') {

in.get(); // ignore '/'

in >> denom; // get b

}

}

void RationalNumber::parse(istream& in, ostream& out) {

bool hasRhs;

RationalNumber rhs;

string op;

in >> \*this; // All input starts with a rational number, as per specs.

in >> op; // All input then as an op, whether it's X, G, +, etc.

// Figure out if it's a binary or unary operation

switch (op[0]) {

// If it's a unary operator...

case 'I':

case 'M':

case 'R':

case 'G':

hasRhs = false;

break;

default:

hasRhs = true;

break;

}

if (hasRhs) { // BINARY OPERATIONS

if (op == "X") {

// Parsing a number of the form (a/b)x+c/d=e/f

char dummy; // For the '+' and '='

RationalNumber cd, ef;

in >> dummy >> cd >> dummy >> ef;

// cout << "DEBUG: Working with: " << \*this << " " << cd << " " << ef

// << flush;

// Solving on paper, I get:

// ((e/f)-(c/d))/(a/b)

// so let's just use that exactly.

// (ef - cd) / ab, reduced to lowest terms.

\*this = ef.sub(cd).div(\*this).reduced();

out << "X = " << \*this;

} else { // Normal binary ops.

in >> rhs;

// Reducing the arthmetic ops' results, as per instructions.

if (op == "+")

\*this = add(rhs).reduced();

else if (op == "-")

\*this = sub(rhs).reduced();

else if (op == "\*")

\*this = mult(rhs).reduced();

else if (op == "/")

\*this = div(rhs).reduced();

else if (op == "L")

setAB(lowestCommonDenominator(rhs), 1);

else if (op == "<")

setAB(isLessThan(rhs), 1);

else if (op == "<=")

setAB(isLessThanEqual(rhs), 1);

else if (op == ">")

setAB(isGreaterThan(rhs), 1);

else if (op == ">=")

setAB(isGreaterThanEqual(rhs), 1);

else if (op == "=")

setAB(isEqual(rhs), 1);

out << \*this;

}

} else { // Unary operations

if (op == "M") {

out << mixedForm();

} else {

if (op == "I")

\*this = invert();

else if (op == "G")

\*this = greatestCommonFactor();

else if (op == "R")

\*this = reduced();

out << \*this;

}

}

}

void RationalNumber::display(std::ostream& out) const {

if (denom == 1) // If we're already an integer...

out << num;

else

out << num << "/" << denom;

}

// OPERATIONS

RationalNumber RationalNumber::invert() const {

return RationalNumber(denom, num);

}

int RationalNumber::greatestCommonFactor() const {

int gcd = 1; // assume they have no GCD other than 1

for (int i = 2; i < max(num, denom); i++) {

if ((num % i == 0) && (denom % i == 0)) // if both divisible by it...

gcd = i; // ....it must be a divisor.

}

return gcd;

}

RationalNumber RationalNumber::reduced() const {

int gcd = greatestCommonFactor();

return RationalNumber(num / gcd, denom / gcd);

}

string RationalNumber::mixedForm() const {

stringstream stream;

stream << num / denom << " + " << num % denom << "/" << denom;

return stream.str();

}

RationalNumber RationalNumber::add(const RationalNumber& rhs) const {

int lcd = lowestCommonDenominator(rhs);

return RationalNumber((lcd / denom) \* num + (lcd / rhs.denom) \* rhs.num,

lcd);

}

RationalNumber RationalNumber::sub(const RationalNumber& rhs) const {

return add(rhs.mult(RationalNumber(-1))); // Addition=sub of opposite.

}

RationalNumber RationalNumber::mult(const RationalNumber& rhs) const {

return RationalNumber(num \* rhs.num, denom \* rhs.denom);

}

RationalNumber RationalNumber::div(const RationalNumber& rhs) const {

return mult(rhs.invert()); // Division is multiplication of the inverted.

}

int RationalNumber::lowestCommonDenominator(const RationalNumber& rhs) const {

if (denom == rhs.denom) {

return denom;

} else {

RationalNumber temp(denom, rhs.denom);

// Multiplying 2 numbers together is the same as putting all their

// factors in a long chain:

// 4\*6 = 2\*2\*2\*3.

// If you divide that by the largest number they have in common

// (GCD): 2\*2\*2\*3 / 2 = 2\*2\*3=12 Gives a number which is the

// multiplication of everything they don't have in common without

// everything they do. (the LCD)

// Or where A,B are sets of the factors: AuB - A(intersection)B

// (Side note: Huh, discrete actually came in handy)

int lcd = denom \* rhs.denom / temp.greatestCommonFactor();

return lcd;

}

}

bool RationalNumber::isLessThan(const RationalNumber& rhs) const {

return sub(rhs).num < 0; // if a-b < 0, then b > a

}

bool RationalNumber::isLessThanEqual(const RationalNumber& rhs) const {

return sub(rhs).num <= 0; // if a-b == 0, they are equal. See above

}

bool RationalNumber::isGreaterThan(const RationalNumber& rhs) const {

return sub(rhs).num > 0; // if a-b > 0, then a > b;

}

bool RationalNumber::isGreaterThanEqual(const RationalNumber& rhs) const {

return sub(rhs).num >= 0;

}

bool RationalNumber::isEqual(const RationalNumber& rhs) const {

RationalNumber reducedThis = reduced(), reducedRhs = rhs.reduced();

return (reducedThis.num == reducedRhs.num) &&

(reducedThis.denom == reducedRhs.denom);

}

// GETTERS AND SETTERS

int RationalNumber::getNum() const {

return num;

}

void RationalNumber::setNum(int value) {

num = value;

}

int RationalNumber::getDenom() const {

return denom;

}

void RationalNumber::setDenom(int value) {

if (value != 0) {

if (value >= 1) {

denom = value;

} else {

num \*= -1;

denom = abs(value);

}

} else

denom = 1;

}

void RationalNumber::setAB(int valueA, int valueB) {

num = valueA;

denom = valueB;

}

ostream& *operator*<<(ostream& out, const RationalNumber& num) {

num.display(*out*);

return out;

}

istream& *operator*>>(istream& in, RationalNumber& num) {

num.read(*in*);

return in;

}

RationalNumber *operator*+(const RationalNumber& lhs, const RationalNumber& rhs) {

return lhs.add(rhs);

}

RationalNumber *operator*-(const RationalNumber& lhs, const RationalNumber& rhs) {

return lhs.sub(rhs);

}

RationalNumber *operator*\*(const RationalNumber& lhs, const RationalNumber& rhs) {

return lhs.mult(rhs);

}

RationalNumber *operator*/(const RationalNumber& lhs, const RationalNumber& rhs) {

return lhs.div(rhs);

}

bool *operator*<(const RationalNumber& lhs, const RationalNumber& rhs) {

return lhs.isLessThan(rhs);

}

bool *operator*<=(const RationalNumber& lhs, const RationalNumber& rhs) {

return lhs.isLessThanEqual(rhs);

}

bool *operator*>(const RationalNumber& lhs, const RationalNumber& rhs) {

return lhs.isGreaterThan(rhs);

}

bool *operator*>=(const RationalNumber& lhs, const RationalNumber& rhs) {

return lhs.isGreaterThanEqual(rhs);

}

bool *operator*==(const RationalNumber& lhs, const RationalNumber& rhs) {

return lhs.isEqual(rhs);

}