#pragma once

#include"fraction.h"

template<typename T>fraction<T>& fraction<T>::reduce()

{

/\*if (denominator == 0)return fraction < T>(1);

//

//std::max(denominator, numerator);

//std::min(denominator, numerator);\*/

T M = std::max(denominator, numerator);

T m = std::min(denominator, numerator);

T gcd = findGCD(M,m);

if (numerator % gcd == 0 && denominator % gcd == 0) {

numerator = (numerator / gcd);

denominator = (denominator / gcd);

}

return \*this;

}

template<typename T>T fraction<T>::findGCD(T& a, T& b)const

{

if (b == 0) {

return a;

}

else {

T c = a % b;

return findGCD(b, c);

}

}

template<typename T>std::ostream& operator<<(std::ostream& os, const fraction<T>& fraction)

{

os << "(" << fraction.numerator << ") / (" << fraction.denominator << ")";

return os;

}

template<typename T>std::istream& operator>>(std::istream& is, fraction<T>& fraction)

{

is >> fraction.numerator;

std::cout << "/\n";

is >> fraction.denominator;

fraction.reduce();

return is;

}

template<typename T>fraction<T> fraction<T>::operator+(const fraction<T>& other)const

{

fraction<T> result;

result.numerator = numerator \* other.denominator + other.numerator \* denominator;

result.denominator = denominator \* other.denominator;

return result.reduce();

}

template<typename T>fraction<T> fraction<T>::operator\*(const fraction<T>& other)const

{

fraction<T> result;

result.numerator = numerator \* other.numerator;

result.denominator = denominator \* other.denominator;

return result.reduce();

}

template<typename T>fraction<T> fraction<T>::operator\*(const T& other)const{

fraction<T> result;

result.numerator = numerator \* other;

result.denominator = denominator;

return result.reduce();

}

template<typename T>fraction<T> fraction<T>::operator/(const fraction<T>& other) const

{

fraction<T> result;

result.numerator = this->numerator \* other.denominator;

result.denominator = this->denominator \* other.numerator;

return result.reduce();

}

template<typename T>fraction<T> fraction<T>::operator-(const fraction<T>& other) const

{

fraction<T> result;

result.numerator = (this->numerator \* other.denominator) - (other.numerator \* this->denominator);

result.denominator = this->denominator \* other.denominator;

return result.reduce();

}

template<typename T>fraction<T> fraction<T>::operator-()const{

fraction<T> result;

result.numerator = (-1)\*(this->numerator);

result.denominator = this->denominator;

return result.reduce();

}

template<typename T>fraction<T> fraction<T>::operator=(const fraction other) {

this->numerator = other.numerator;

this->denominator = other.denominator;

return \*this;

}

template<typename T>fraction<T> fraction<T>::operator=(const T other) {

numerator = other;

denominator = 1;

return \*this;

}

template<typename T>bool fraction<T>::operator==(const fraction& other) const {

return (this->numerator \* other.denominator) == (other.numerator \* this->denominator);

}

template<typename T>bool fraction<T>::operator!=(const fraction<T>& other) const {

return !(\*this == other);

}

template<typename T>bool fraction<T>::operator>(const fraction& other) const {

return (numerator \* other.denominator) > (other.numerator \* denominator);

}

template<typename T>bool fraction<T>::operator<(const fraction& other) const {

return (numerator \* other.denominator) < (other.numerator \* denominator);

}

template<typename T>bool fraction<T>::operator==(const T other) const {

if (other == 0 && numerator == 0) {

return true;

}

else {

if (fraction<T>(1) ==(\*this))

{

return true;

}

}

return false;

}