Experiment #2

1.

//Complex.h

#ifndef Complex\_h

#define Complex\_h

class Complex {

//friend Complex operator = (const Complex&);

friend Complex operator + (const Complex&, const Complex&);

friend Complex operator - (const Complex&, const Complex&);

friend Complex operator \* (const Complex&, const Complex&);

friend bool operator == (const Complex&, const Complex&);

friend bool operator != (const Complex&, const Complex&);

friend ostream& operator <<(ostream&, const Complex&);

friend istream& operator >>(istream&, Complex&);

public:

Complex(double = 0);

Complex(double, double);

Complex(double[2]);

private:

double char\_double(char[], int, int, int);

double cplx[2];

};

#endif // !Complex\_h

//////////////////////////////////////////////////

//Complex.cpp

#include<iostream>

#include<cmath>

using namespace std;

#include"Complex.h"

Complex operator + (const Complex& c1, const Complex& c2) {

Complex result;

result.cplx[0] = c1.cplx[0] + c2.cplx[0];

result.cplx[1] = c1.cplx[1] + c2.cplx[1];

return result;

}//( a + bi ) + ( c + di )

Complex operator - (const Complex& c1, const Complex& c2) {

Complex result;

result.cplx[0] = c1.cplx[0] - c2.cplx[0];

result.cplx[1] = c1.cplx[1] - c2.cplx[1];

return result;

}//( a + bi ) - ( c + di )

Complex operator \* (const Complex& c1, const Complex& c2) {

Complex result;

result.cplx[0] = c1.cplx[0] \* c2.cplx[0] - c1.cplx[1] \* c2.cplx[1];

result.cplx[1] = c1.cplx[0] \* c2.cplx[1] + c2.cplx[0] \* c1.cplx[1];

return result;

}//( a + bi ) \* ( c + di )

bool operator == (const Complex& c1, const Complex& c2) {

if (c1.cplx[0] == c2.cplx[0] && c1.cplx[1] == c2.cplx[1])

return true;

else

return false;

}

bool operator != (const Complex& c1, const Complex& c2) {

if (c1 == c2)

return false;

else

return true;

}

ostream& operator <<(ostream& output, const Complex& c) {

cout << "( " << c.cplx[0] << " + " << c.cplx[1] << "i )";

return output;

}//cout << ( a + bi )

istream& operator >>(istream& input, Complex& c) {

const int size = 50;

char char\_input[size];

cout << "Input a complex number in the form: ( realPart , imaginaryPart )\n";

cin.getline(char\_input, size);//input like ( a , b )

int index\_begin = 0, index\_point = 0, index\_end = 0;

int judger = 0;//to decide input rPart or iPart

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

if (char\_input[i] == ',')

judger++;

if (char\_input[i] >= '0' && char\_input[i] <= '9') {

index\_begin = i;//mark the first number appearing in "char\_input"

for (int j = 1; j < size - index\_begin; j++) {

char temp = char\_input[index\_begin + j];

index\_end = index\_begin + j - 1;

if ((temp >= '0' && temp <= '9') || temp == '.')

index\_end++;

else

break;//mark the end, and break the loop when no number in the next

if (temp == '.')

index\_point = index\_begin + j;//mark the "." to seperate the integer and demical parts

}

if (index\_point == 0)

index\_point = index\_end + 1;//in case of it is an integer

c.cplx[judger] = c.char\_double(char\_input, index\_begin, index\_point, index\_end);//realPart or imaginaryPart

if (index\_begin > 0 && char\_input[index\_begin - 1] == '-')

c.cplx[judger] = -c.cplx[judger];

i = index\_end + 1;

index\_begin = 0;

index\_point = 0;

index\_end = 0;//reset all index to 0

}

if (judger >= 2)

break;

}

return input;

}//cin >> ( a , b )

double Complex::char\_double(char char\_input[], int index\_begin, int index\_point, int index\_end) {

double result = 0;

for (int i = index\_begin; i <= index\_end; i++) {

if (i < index\_point)

result += (char\_input[i] - '0') \* pow(10, index\_point - i - 1);

else if (i > index\_point)

result += (char\_input[i] - '0') \* pow(10, index\_point - i);

}

return result;

}//collect valid number in "char\_input", and make it a double number

Complex::Complex(double c) {

cplx[0] = c;

cplx[1] = 0;

}//constructor

Complex::Complex(double r, double i) {

cplx[0] = r;

cplx[1] = i;

}//constructor

Complex::Complex(double c[2]) {

cplx[0] = c[0];

cplx[1] = c[1];

}//constructor

//////////////////////////////////////////////////

//main.cpp

#include<iostream>

using namespace std;

#include"Complex.h"

int main() {

double array[2] = { 3.3 , 1.1 };

Complex x;//default: 0 + 0i

Complex y(4.3, 8.2);//constructor\_1: 4.3 + 8.2i

Complex z(array);//constructor\_2: 3.3 + 1.1i

Complex k;

cin >> k;

cout << "\nx: " << x << "\ny: " << y << "\nz: " << z << "\nk: " << k;

cout << endl << endl;

cout << "x = y + z:\n";

x = y + z;

cout << x << " = " << y << " + " << z << endl << endl;

cout << "x = y - z:\n";

x = y - z;

cout << x << " = " << y << " - " << z << endl << endl;

cout << "x = y \* z:\n";

x = y \* z;

cout << x << " = " << y << " \* " << z << endl << endl;

if (x != k)

cout << x << " != " << k << endl << endl;

x = k;

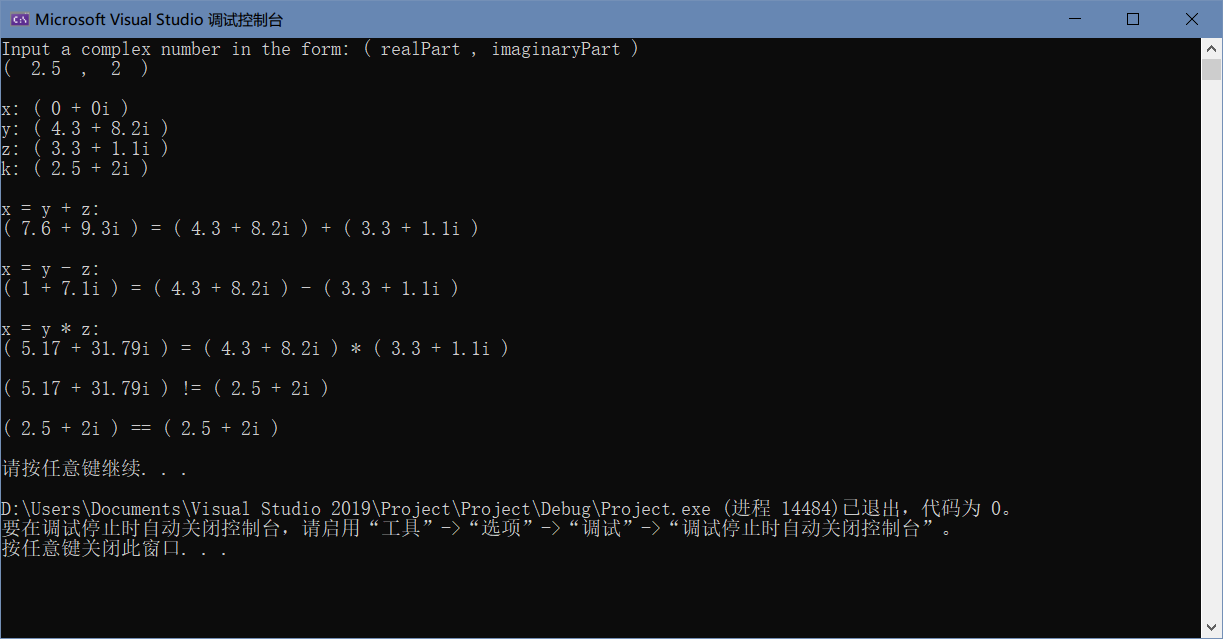
if (x == k)

cout << x << " == " << k << endl << endl;

system("pause");

return 0;

}



2.

//HugeInteger.h

#ifndef HugeInteger\_h

#define HugeInteger\_h

class HugeInteger{

friend ostream& operator<<(ostream&, const HugeInteger&);

friend istream& operator>>(istream&, HugeInteger&);

public:

HugeInteger(int = 0);

HugeInteger(const char\*, int);

HugeInteger operator+(const HugeInteger&)const;

HugeInteger operator-(const HugeInteger&)const;

HugeInteger operator+=(const HugeInteger&);

HugeInteger operator-=(const HugeInteger&);

HugeInteger operator\*(const HugeInteger&)const;

HugeInteger operator/(const HugeInteger&)const;

bool operator==(const HugeInteger&)const;

bool operator!=(const HugeInteger&)const;

bool operator>(const HugeInteger&)const;

bool operator<(const HugeInteger&)const;

bool operator>=(const HugeInteger&)const;

bool operator<=(const HugeInteger&)const;

private:

int integer[40];

};

#endif // !HugeInteger\_h

//////////////////////////////////////////////////

//HugeInteger.cpp

#include<iostream>

#include<cmath>

#include<string>

using namespace std;

#include"HugeInteger.h"

ostream& operator<<(ostream& output, const HugeInteger& HI) {

bool judger = 0;

int counter = 0;

for (int i = 0; i < 40; i++) {

if (judger || HI.integer[i] != 0) {

cout << HI.integer[i];

judger = 1;

}

else {

judger = 0;

counter++;

}

}

if (counter == 40)

cout << 0;

return output;

}//cout << "HugeInteger"

istream& operator>>(istream& input, HugeInteger& HI) {

char num\_c[40];

cin.getline(num\_c, 40);

int size = 0;

if (num\_c[0] < '0' || num\_c[0] > '9') {

num\_c[0] = '0';

num\_c[1] = '\0';

}

for (int i = 0; i < 40; i++) {

if (num\_c[i] == '\0')

break;

else

size++;

}//get valid size

if (size == 0)

size++;

for (int i = 39; i >= 0; i--)

if (i >= 40 - size)

HI.integer[i] = num\_c[size - 40 + i] - '0';

else

HI.integer[i] = 0;

return input;

}//cin >> "HugeInteger"

HugeInteger::HugeInteger(int num\_i) {

int figure = 0;

if (num\_i) {

for (int i = 0; i < 41; i++) {

if (num\_i / pow(10, i) >= 1)

figure++;

else

break;

}

}

else

figure = 1;

//Judge how many figures num\_i has

for (int i = 39; i >= 0; i--) {

if (i > 39 - figure) {

int temp = num\_i / pow(10, 40 - i);

integer[i] = num\_i / pow(10, 39 - i) - temp \* 10;

}

else

integer[i] = 0;

}

//Convert num\_i into a HugeInteger number

} // conversion/default constructor

HugeInteger::HugeInteger(const char\* num\_c, int size) {

for (int i = 39; i >= 0; i--) {

if (i >= 41 - size)

integer[i] = \*(num\_c + size - 41 + i) - '0';

else

integer[i] = 0;

}

} // conversion constructor (I add a parameter "size" to the constructor)

HugeInteger HugeInteger::operator+(const HugeInteger& HI)const {

HugeInteger result;

int carry = 0;

for (int i = 39; i >= 0; --i) {

int temp = integer[i] + HI.integer[i] + carry;

if (temp > 9) {

result.integer[i] = temp - 10;

carry = 1;

}

else {

result.integer[i] = temp;

carry = 0;

}//Judge whether to carry

}

if (carry) {

cout << "\nDATA OVERFLOW!\nThe figures of result is more than 40!\n";

return 0;

}

return result;

}//addition

HugeInteger HugeInteger::operator-(const HugeInteger& HI)const {

HugeInteger result;

int borrow = 0;

for (int i = 39; i >= 0; --i) {

int temp = integer[i] - HI.integer[i] - borrow;

if (temp < 0) {

result.integer[i] = temp + 10;

borrow = 1;

}

else {

result.integer[i] = temp;

borrow = 0;

}//Judge whether to borrow

}

if (borrow) {

cout << "\nNEGATIVE RESULT!\nThe minuend is less than the subtracted!\n";

return 0;

}

return result;

}//sutraction

HugeInteger HugeInteger::operator+=(const HugeInteger& HI) {

\*this = \*this + HI;

return \*this;

}

HugeInteger HugeInteger::operator-=(const HugeInteger& HI) {

\*this = \*this - HI;

return \*this;

}

HugeInteger HugeInteger::operator\*(const HugeInteger& HI)const {

HugeInteger result;

HugeInteger temp = HI;

while (temp > 0) {

result += \*this;

temp -= 1;

}

return result;

}//multiplication

HugeInteger HugeInteger::operator/(const HugeInteger& HI)const {

HugeInteger result;

HugeInteger temp = \*this;

while (temp > HI) {

result += 1;

temp -= HI;

}

return result;

}//division

bool HugeInteger::operator==(const HugeInteger& HI)const {

bool judger = 1;

for (int i = 0; i < 40; i++)

if (integer[i] != HI.integer[i])

judger = 0;

if (judger)

return 1;

else

return 0;

}//equality

bool HugeInteger::operator!=(const HugeInteger& HI)const {

if (\*this == HI)

return 0;

else

return 1;

}//inequality

bool HugeInteger::operator>(const HugeInteger& HI)const {

if (\*this != HI) {

int borrow = 0;

for (int i = 39; i >= 0; --i) {

int temp = integer[i] - HI.integer[i] - borrow;

if (temp < 0)

borrow = 1;

else

borrow = 0;

}

if (borrow)

return 0;

else

return 1;

}

else

return 0;

}//greater than

bool HugeInteger::operator<(const HugeInteger& HI)const {

if (\*this == HI || \*this > HI)

return 0;

else

return 1;

}//less than

bool HugeInteger::operator>=(const HugeInteger& HI)const {

if (\*this < HI)

return 0;

else

return 1;

}

bool HugeInteger::operator<=(const HugeInteger& HI)const {

if (\*this > HI)

return 0;

else

return 1;

}

//////////////////////////////////////////////////

//main.cpp

#include<iostream>

#include<cmath>

#include<string>

using namespace std;

#include"HugeInteger.h"

int main() {

int n1\_i = 7654321, n2\_i = 7891234;

char n3\_c[] = "99999999999999999999", n4\_c[] = "1";

HugeInteger n1(n1\_i);

HugeInteger n2(n2\_i);

HugeInteger n3(n3\_c, sizeof(n3\_c));

HugeInteger n4(n4\_c, sizeof(n4\_c));

HugeInteger n5(12341234);

HugeInteger n6;

cout << "n1 is: " << n1 << endl;

cout << "n2 is: " << n2 << endl;

cout << "n3 is: " << n3 << endl;

cout << "n4 is: " << n4 << endl;

cout << "n5 is: " << n5 << endl;

cout << "Input n6: ";

cin >> n6;

cout << "n6 is: " << n6 << endl << endl;

cout << "n1 is" << ((n1 == n2) ? " " : " not ") << "equal to n2\n";

cout << "n1 is" << ((n1 < n2) ? " " : " not ") << "less than n2\n";

cout << "n1 is" << ((n1 <= n2) ? " " : " not ") << "less than or equal to n2\n\n";

cout << n1 << " + " << n2 << " = " << (n1 + n2) << endl;

cout << n3 << " + " << n4 << " = " << (n3 + n4) << endl << endl;

cout << n1 << " + 9 = " << (n1 + 9) << endl;

cout << n2 << " + 10000 = " << (n2 + 10000) << endl;

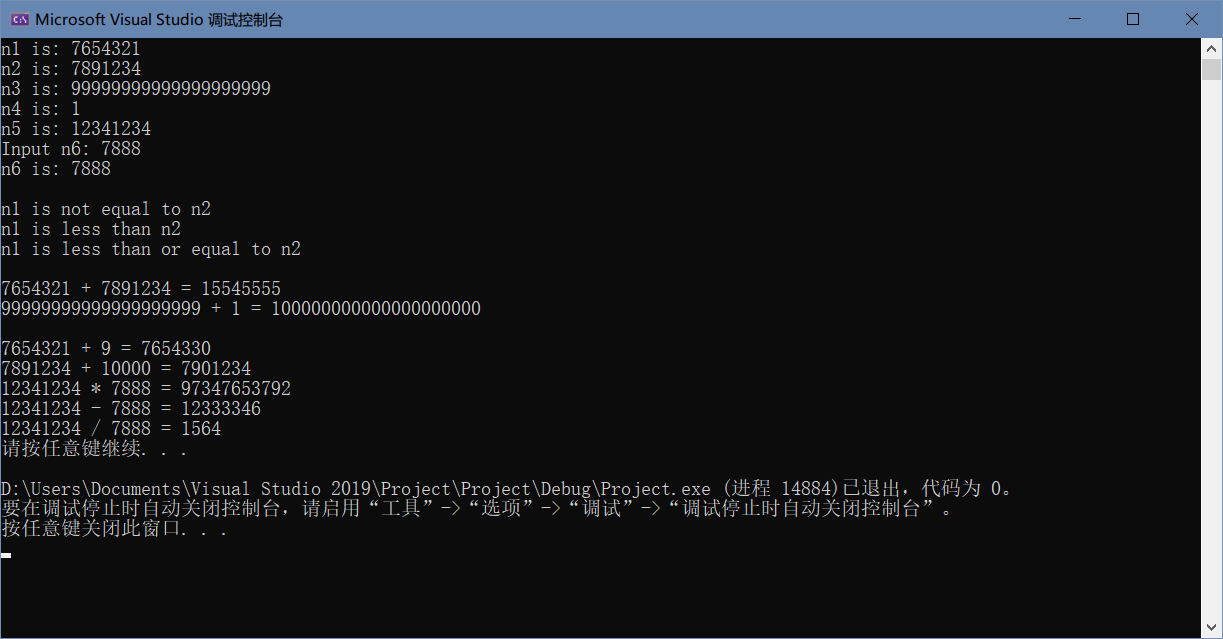
cout << n5 << " \* " << n6 << " = " << (n5 \* n6) << endl;

cout << n5 << " - " << n6 << " = " << (n5 - n6) << endl;

cout << n5 << " / " << n6 << " = " << (n5 / n6) << endl; system("pause");

return 0;

}



3.

//RationalNumber.h

#ifndef RationalNumber\_h

#define RationalNumber\_h

int GCD(int, int);

int LCM(int, int);

class RationalNumber{

public:

RationalNumber(int = 0, int = 1); // default constructor

void Input();

RationalNumber operator+(const RationalNumber&) const; // addition

RationalNumber operator-(const RationalNumber&) const; // subtraction

RationalNumber operator\*(const RationalNumber&) const; // multiplication

RationalNumber operator/(const RationalNumber&) const; // division

// relational operators

bool operator>(const RationalNumber&) const;

bool operator<(const RationalNumber&) const;

bool operator>=(const RationalNumber&) const;

bool operator<=(const RationalNumber&) const;

// equality operators

bool operator==(const RationalNumber&) const;

bool operator!=(const RationalNumber&) const;

void printRational(); // display rational number

private:

int numerator; // private variable numerator

int denominator; // private variable denominator

void reduction(); // function for fraction reduction

};

#endif // !RationalNumber\_h

//////////////////////////////////////////////////

//RationalNumber.cpp

#include<iostream>

using namespace std;

#include"RationalNumber.h"

int GCD(int a, int b){

int temp = 0;

if (a < b) {

temp = a;

a = b;

b = temp;

}

while (b != 0) {

temp = a % b;

a = b;

b = temp;

}

return a;

}//greatest common divisor

int LCM(int a, int b){

int t = a \* b / GCD(a, b);

return t;

}//least common multiple

void RationalNumber::reduction() {

int gcd = 0;

if (numerator < 0)

gcd = GCD(-numerator, denominator);

else

gcd = GCD(numerator, denominator);//to get a posivie greatest common divisor

if (gcd > 1) {

numerator /= gcd;

denominator /= gcd;

}

}// function for fraction reduction

RationalNumber::RationalNumber(int n, int d) {

if (d <= 0) {

cout << "The denominator should be a POSITIVE INTEGER.\nPlease input again.\n";

Input();

}

else {

numerator = n;

denominator = d;

reduction();

}

}//default constructor

void RationalNumber::Input() {

cout << "The numerator is: ";

cin >> numerator;

cout << "The denominator is: ";

cin >> denominator;

if (denominator <= 0) {

cout << "The denominator should be a positive integer.\nPlease input again.\n";

Input();

}

reduction();

}

RationalNumber RationalNumber::operator+(const RationalNumber& RN) const {

RationalNumber result;

if (denominator == RN.denominator) {

result.denominator = denominator;

result.numerator = numerator + RN.numerator;

}

else {

int lcm = LCM(denominator, RN.denominator);

result.denominator = lcm;

result.numerator = numerator \* lcm / denominator + RN.numerator \* lcm / RN.denominator;

}

result.reduction();

return result;

}//addition

RationalNumber RationalNumber::operator-(const RationalNumber& RN) const {

RationalNumber result;

if (denominator == RN.denominator) {

result.denominator = denominator;

result.numerator = numerator - RN.numerator;

}

else {

int lcm = LCM(denominator, RN.denominator);

result.denominator = lcm;

result.numerator = numerator \* lcm / denominator - RN.numerator \* lcm / RN.denominator;

}

result.reduction();

return result;

}//subtraction

RationalNumber RationalNumber::operator\*(const RationalNumber& RN) const {

RationalNumber result;

result.denominator = denominator \* RN.denominator;

result.numerator = numerator \* RN.numerator;

result.reduction();

return result;

}//multiplication

RationalNumber RationalNumber::operator/(const RationalNumber& RN) const {

if (RN.numerator == 0) {

cout << "INVALID\nThe divisor can\'t be 0.\n";

return 0;

}

else {

RationalNumber reciprocal;

reciprocal.denominator = RN.numerator;

reciprocal.numerator = RN.denominator;

return (\*this \* reciprocal);

}

}//division

// relational operators

bool RationalNumber::operator>(const RationalNumber& RN) const {

if ((\*this - RN).numerator > 0)

return 1;

else

return 0;

}

bool RationalNumber::operator<(const RationalNumber& RN) const {

if ((\*this - RN).numerator < 0)

return 1;

else

return 0;

}

bool RationalNumber::operator>=(const RationalNumber& RN) const {

if ((\*this - RN).numerator >= 0)

return 1;

else

return 0;

}

bool RationalNumber::operator<=(const RationalNumber& RN) const {

if ((\*this - RN).numerator <= 0)

return 1;

else

return 0;

}

// equality operators

bool RationalNumber::operator==(const RationalNumber& RN) const {

if ((\*this - RN).numerator == 0)

return 1;

else

return 0;

}

bool RationalNumber::operator!=(const RationalNumber& RN) const {

if ((\*this - RN).numerator != 0)

return 1;

else

return 0;

}

void RationalNumber::printRational() {

if (denominator == 1)

cout << numerator;

else

cout << numerator << '/' << denominator;

}//display rational number

//////////////////////////////////////////////////

//main.cpp

#include<iostream>

using namespace std;

#include"RationalNumber.h"

int main() {

RationalNumber c(7, 3), d(3, 9), x;

c.printRational();

cout << " + ";

d.printRational();

cout << " = ";

x = c + d;//test overloaded operators + and =

x.printRational();

cout << '\n';

c.printRational();

cout << " - ";

d.printRational();

cout << " = ";

x = c - d;//test overloaded operators - and =

x.printRational();

cout << '\n';

c.printRational();

cout << " \* ";

d.printRational();

cout << " = ";

x = c \* d;//test overloaded operators \* and =

x.printRational();

cout << '\n';

c.printRational();

cout << " / ";

d.printRational();

cout << " = ";

x = c / d; // test overloaded operators / and =

x.printRational();

cout << '\n';

c.printRational();

cout << " is:\n";

// test overloaded greater than operator

cout << ((c > d) ? " > " : " <= ");

d.printRational();

cout << " according to the overloaded > operator\n";

// test overloaded less than operator

cout << ((c < d) ? " < " : " >= ");

d.printRational();

cout << " according to the overloaded < operator\n";

// test overloaded greater than or equal to operator

cout << ((c >= d) ? " >= " : " < ");

d.printRational();

cout << " according to the overloaded >= operator\n";

// test overloaded less than or equal to operator

cout << ((c <= d) ? " <= " : " > ");

d.printRational();

cout << " according to the overloaded <= operator\n";

// test overloaded equality operator

cout << ((c == d) ? " == " : " != ");

d.printRational();

cout << " according to the overloaded == operator\n";

// test overloaded inequality operator

cout << ((c != d) ? " != " : " == ");

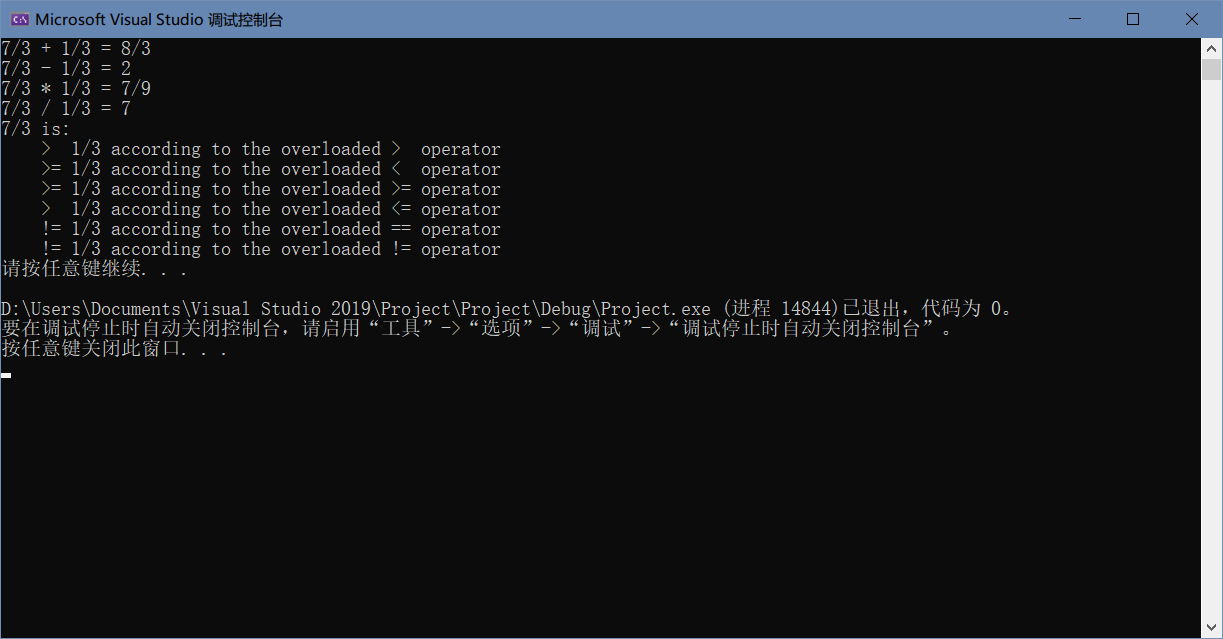
d.printRational();

cout << " according to the overloaded != operator" << endl;

system("pause");

return 0;

}



4.

//String.h

#ifndef String\_h

#define String\_h

class String {

friend ostream& operator<<(ostream&, const String&);

public:

String(const char\* const = " "); // conversion constructor

String(const String&); // copy constructor

~String(); // destructor

const String& operator=(const String&);

String operator+(const String&);

private:

char\* sPtr; // pointer to start of string

int length; // string length

}; // end class String

#endif // !String\_h

//////////////////////////////////////////////////

//String.cpp

#include <iostream>

#include <cstring>

#include <cassert>

using namespace std;

#include"String.h"

ostream& operator<<(ostream& output, const String& s) {

for (int i = 0; i < s.length; i++)

cout << \*(s.sPtr + i);

return output;

}

String::String(const char\* const c) {

length = 0;

for (int i = 0; i < 100; i++) {

if (\*(c + i) == '\0')

break;

else

length++;

}

sPtr = new char[length];

for (int i = 0; i < length; i++)

\*(sPtr + i) = \*(c + i);

}//convesion constructor

String::String(const String& s) {

length = s.length;

sPtr = new char[length];

for (int i = 0; i < length; i++)

\*(sPtr + i) = \*(s.sPtr + i);

}//copy constructor

String::~String() {

delete[] sPtr;

sPtr = nullptr;

}//destructor

const String& String::operator=(const String& s) {

if (sPtr == s.sPtr)

return \*this;

else {

length = s.length;

delete[] sPtr;

sPtr = new char[length];

for (int i = 0; i < length; i++)

\*(sPtr + i) = \*(s.sPtr + i);

return \*this;

}

}

String String::operator+(const String& s) {

String temp;

temp.length = length + s.length;

temp.sPtr = new char[length + s.length];

for (int i = 0; i < length + s.length; i++) {

if (i < length)

\*(temp.sPtr + i) = \*(sPtr + i);

else

\*(temp.sPtr + i) = \*(s.sPtr + i - length);

}

return temp;

}

//////////////////////////////////////////////////

//main.cpp

#include<iostream>

using namespace std;

#include "String.h"

int main(){

String string1, string2("The date is");

String string3(" August 1, 1993");

// test overloaded operators

cout << "string1 = string2 + string3\n";

string1 = (string2 + string3);// tests overloaded = an d + operator

cout << "\"" << string1 << "\" = \"" << string2 << "\" + \"" << string3 << "\"" << endl;

system("pause");

return 0;

}

