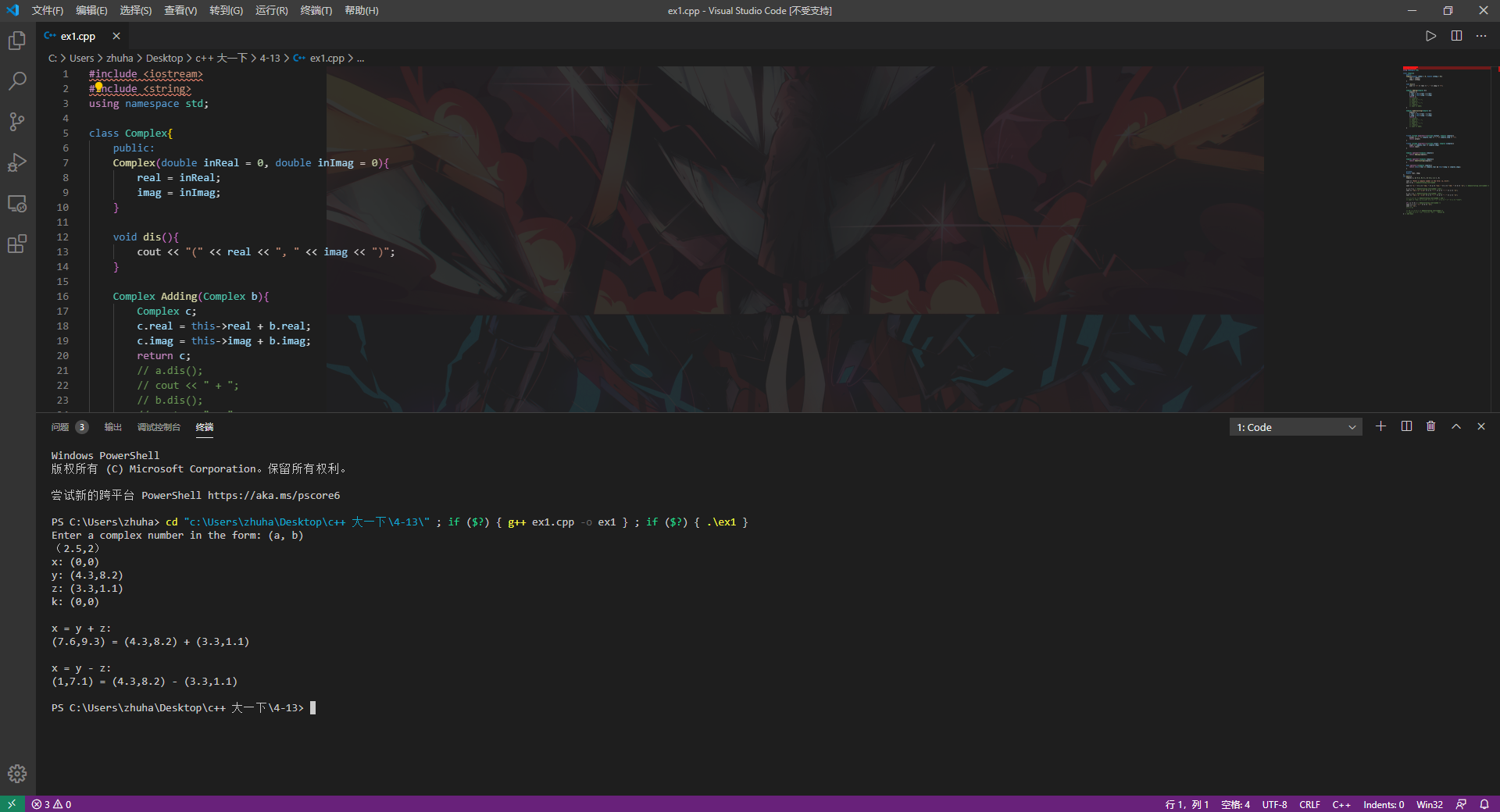
Ex1



#include <iostream>

#include <string>

using namespace std;

class Complex{

    public:

    Complex(double inReal = 0, double inImag = 0){

        real = inReal;

        imag = inImag;

    }

    void dis(){

        cout << "(" << real << ", " << imag << ")";

    }

    Complex Adding(Complex b){

        Complex c;

        c.real = this->real + b.real;

        c.imag = this->imag + b.imag;

        return c;

        // a.dis();

        // cout << " + ";

        // b.dis();

        // cout << " = ";

        // c.dis();

        // cout << endl;

    }

    Complex Substracting(Complex b){

        Complex c;

        c.real = this->real - b.real;

        c.imag = this->imag - b.imag;

        return c;

        // a.dis();

        // cout << " - ";

        // b.dis();

        // cout << " = ";

        // c.dis();

        // cout << endl;

    }

    friend ostream &operator<<(ostream &output, Complex complex){

        output << "(" << complex.real << "," << complex.imag << ")";

        return output;

    }

    friend istream &operator>>(istream &input, Complex &complex){

        input >> complex.real >> complex.imag;

        return input;

    }

    Complex operator+(Complex complex){

        return Adding(complex);

    }

    Complex operator-(Complex complex){

        return Substracting(complex);

    }

    bool operator!=(Complex complex){

        return (this->real == complex.real && this->imag == complex.imag);

    }

    private:

    double real, imag;

};

int main(){

    Complex x, y( 4.3, 8.2 ), z( 3.3, 1.1 ), k;

    cout << "Enter a complex number in the form: (a, b)\n";

    cin >> k; // demonstrating overloaded

    cout << "x: " << x << "\ny: " << y << "\nz: " << z << "\nk: " << k << '\n'; // demonstrating overloaded <<

    x = y + z; // demonstrating overloaded + and =

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

    x = y - z; // demonstrating overloaded - and =

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

    // x = y \* z; // demonstrating overloaded \* and =

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

    if ( x != k ) // demonstrating overloaded !=

    cout << x << " != " << k << '\n';

    cout << '\n';

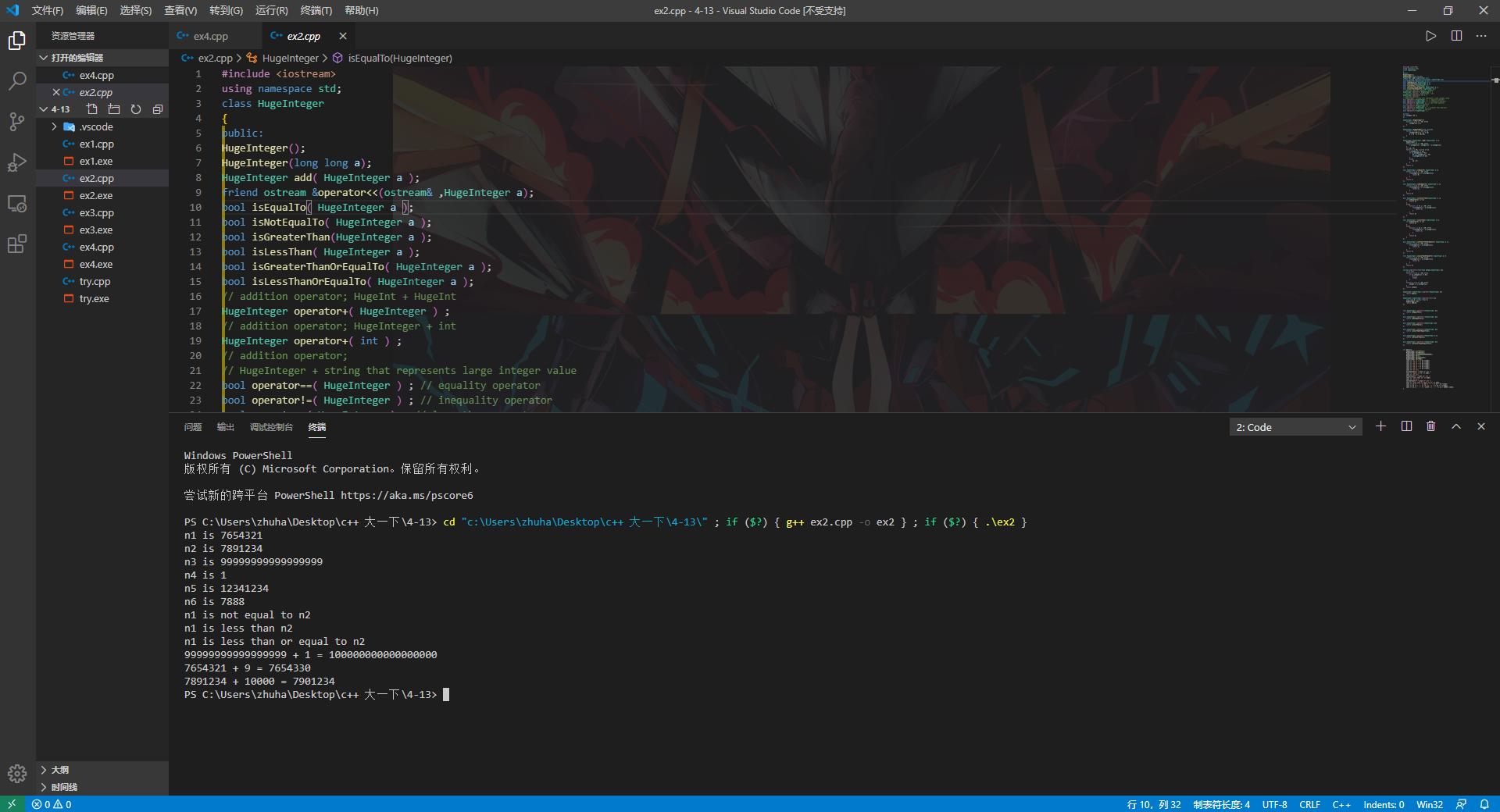
    // x = k;

    // if ( x == k ) // demonstrating overloaded ==

    // cout << x << " == " << k << '\n';    return 0;

} // end main

Ex2



#include <iostream>

using namespace std;

class HugeInteger

{

public:

HugeInteger();

HugeInteger(long long a);

HugeInteger add( HugeInteger a );

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

bool isEqualTo( HugeInteger a );

bool isNotEqualTo( HugeInteger a );

bool isGreaterThan(HugeInteger a );

bool isLessThan( HugeInteger a );

bool isGreaterThanOrEqualTo( HugeInteger a );

bool isLessThanOrEqualTo( HugeInteger a );

// addition operator; HugeInt + HugeInt

HugeInteger operator+( HugeInteger ) ;

// addition operator; HugeInteger + int

HugeInteger operator+( int ) ;

// addition operator;

// HugeInteger + string that represents large integer value

bool operator==( HugeInteger ) ; // equality operator

bool operator!=( HugeInteger ) ; // inequality operator

bool operator<( HugeInteger ) ; // less than operator

// less than or equal to operator

bool operator<=( HugeInteger ) ;

bool operator>( HugeInteger ) ; // greater than operator

// greater than or equal to operator

bool operator>=( HugeInteger ) ;

private:

int integer[ 40 ];

};

HugeInteger::HugeInteger(){

    for(int i = 0; i < 40; i++){

        integer[i] = 0;

    }

};

HugeInteger::HugeInteger(long long a){

    for(int i = 0; i < 40; i++){

        integer[40-i-1] = a % 10;

        a = (a - a % 10)/10;

    }

};

HugeInteger HugeInteger::add( HugeInteger a ){

    HugeInteger c;

    for(int i = 0; i < 40; i++){

        c.integer[i] = integer[i] + a.integer[i];

    }

    int up = 0;

    for(int i = 39; i >= 0; i--){

        c.integer[i] += up;

        if (c.integer[i] > 9){

            up = c.integer[i] / 10;

            c.integer[i] %= 10;

        }

        else{

            up = 0;

        }

    }

    return c;

};

bool HugeInteger::isEqualTo( HugeInteger a ){

    for(int i = 0; i < 40; i++){

        if(integer[i] != a.integer[i]){

            return 0;

        }

    }

    return 1;

};

bool HugeInteger::isNotEqualTo( HugeInteger a ){

    for(int i = 0; i < 40; i++){

        if(integer[i] != a.integer[i]){

            return 1;

        }

    }

    return 0;

};

bool HugeInteger::isGreaterThan(HugeInteger a ){

    if (isEqualTo(a) == 1){

        return 0;

    }

    else{

        for(int i = 0; i < 40; i++){

            if(integer[i] > a.integer[i]){

                return 1;

            }

        }

        return 0;

    }

};

bool HugeInteger::isLessThan( HugeInteger a ){

    if (isEqualTo(a) == 1){

        return 0;

    }

    else{

        for(int i = 0; i < 40; i++){

            if(integer[i] < a.integer[i]){

                return 1;

            }

        }

        return 0;

    }

};

bool HugeInteger::isGreaterThanOrEqualTo( HugeInteger a ){

    for(int i = 0; i < 40; i++){

        if(integer[i] > a.integer[i]){

            return 1;

        }

    }

    return 0;

};

bool HugeInteger::isLessThanOrEqualTo( HugeInteger a ){

    for(int i = 0; i < 40; i++){

        if(integer[i] < a.integer[i]){

            return 1;

        }

    }

    return 0;

};

ostream &operator<<(ostream& output,HugeInteger a){

    int k = 0;

    for(int i = 0; i < 40; i++){

        if (a.integer[i] != 0){

            k = i;

            break;

        }

    }

    for(int j = k; j < 40; j++){

        output << a.integer[j];

    }

    return output;

};

HugeInteger HugeInteger::operator+(HugeInteger a){

    return add(a);

};

HugeInteger HugeInteger::operator+(int a){

    long long b = (long long) a;

    HugeInteger k(b);

    return add(k);

};

bool HugeInteger::operator==(HugeInteger a){

    return isEqualTo(a);

};

bool HugeInteger::operator!=(HugeInteger a){

    return isNotEqualTo(a);

};

bool HugeInteger::operator<(HugeInteger a){

    return isLessThan(a);

};

bool HugeInteger::operator<=(HugeInteger a){

    return isLessThanOrEqualTo(a);

};

bool HugeInteger::operator>(HugeInteger a ){

    return isGreaterThan(a);

};

bool HugeInteger::operator>=(HugeInteger a){

    return isGreaterThanOrEqualTo(a);

};

int main(){

    HugeInteger n1(7654321);

    HugeInteger n2(7891234);

    HugeInteger n3(99999999999999999);

    HugeInteger n4(1);

    HugeInteger n5(12341234);

    HugeInteger n6(7888);

    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 << "n6 is " << n6 <<endl;

    cout << "n1 is";

    n1==n2?cout<<" ":cout <<" not ";

    cout << "equal to n2" << endl;

    cout << "n1 is";

    n1<n2?cout<<" ":cout <<" not ";

    cout << "less than n2" << endl;

    cout << "n1 is";

    n1<=n2?cout<<" ":cout <<" not ";

    cout << "less than or equal to n2" << endl;

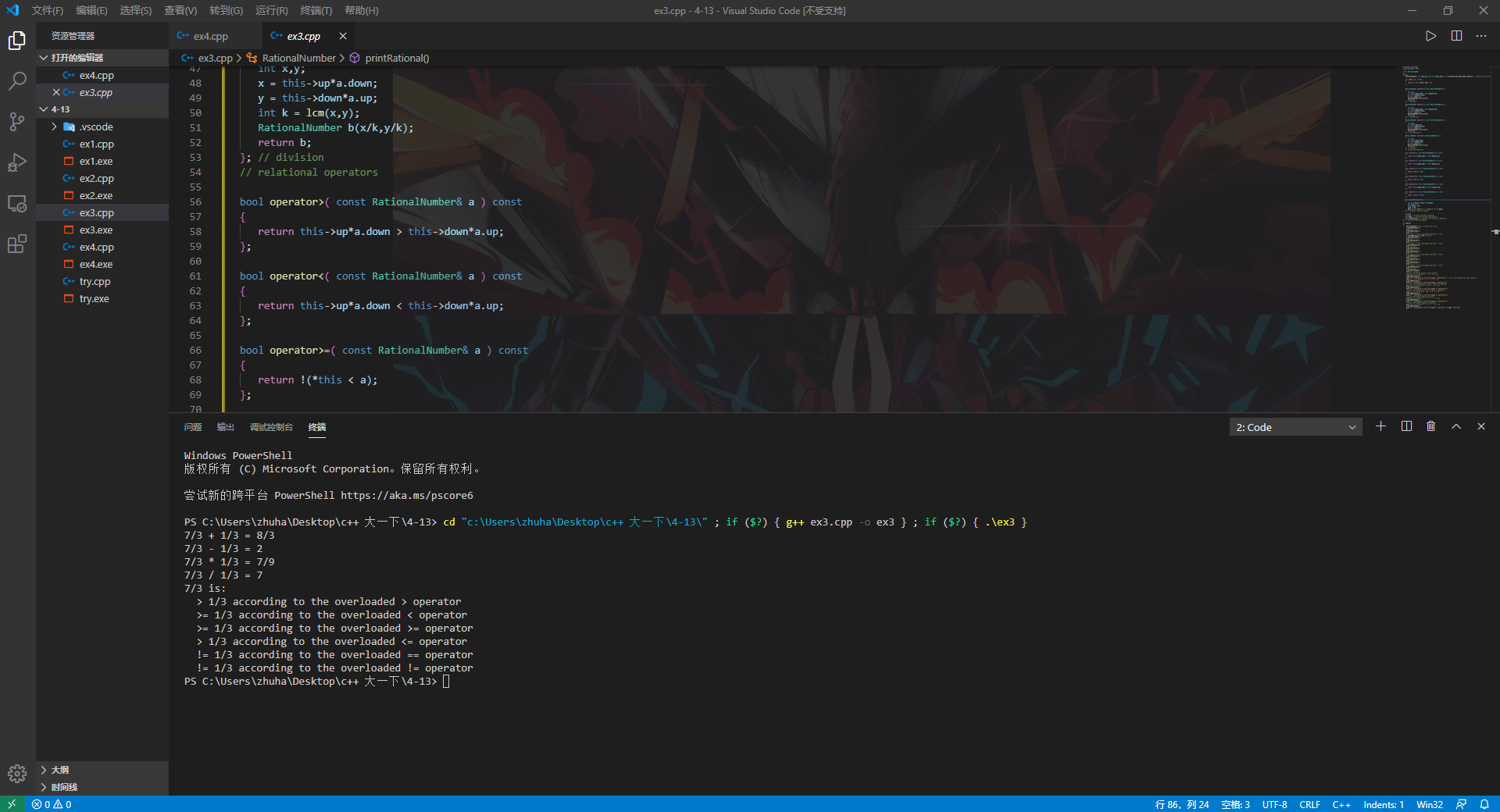
    cout << n3 << " + " << n4 << " = " << n3 + n4 <<endl;

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

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

}

Ex3



#include <iostream>

using namespace std;

class RationalNumber

{

public:

   RationalNumber( int input\_up = 0, int input\_down = 1 ):up(input\_up),down(input\_down){}; // default constructor

   int lcm(int a, int b)

   {

      return b > 0 ? lcm(b, a%b) : a;

   }

   RationalNumber operator+( const RationalNumber& a )

   {

      int x,y;

      x = this->up\*a.down + this->down\*a.up;

      y = this->down\*a.down;

      int k = lcm(x,y);

      RationalNumber b(x/k,y/k);

      return b;

   }; // addition

   RationalNumber operator-( const RationalNumber& a )

   {

      int x,y;

      x = this->up\*a.down - this->down\*a.up;

      y = this->down\*a.down;

      int k = lcm(x,y);

      RationalNumber b(x/k,y/k);

      return b;

   }; // subtraction

   RationalNumber operator\*( const RationalNumber& a )

   {

      int x,y;

      x = this->up\*a.up;

      y = this->down\*a.down;

      int k = lcm(x,y);

      RationalNumber b(x/k,y/k);

      return b;

   }; // multiplication

   RationalNumber operator/( RationalNumber& a )

   {

      int x,y;

      x = this->up\*a.down;

      y = this->down\*a.up;

      int k = lcm(x,y);

      RationalNumber b(x/k,y/k);

      return b;

   }; // division

   // relational operators

   bool operator>( const RationalNumber& a ) const

   {

      return this->up\*a.down > this->down\*a.up;

   };

   bool operator<( const RationalNumber& a ) const

   {

      return this->up\*a.down < this->down\*a.up;

   };

   bool operator>=( const RationalNumber& a ) const

   {

      return !(\*this < a);

   };

   bool operator<=( const RationalNumber& a ) const

   {

      return !(\*this > a);

   };

   bool operator==( const RationalNumber& a ) const

   {

      return this->up\*a.down == this->down\*a.up;

   };

   bool operator!=( const RationalNumber& a ) const

   {

      return !(\*this == a);

   };

   void printRational()

   {

      int k = lcm(this->up,this->down);

      this->up /= k;

      this->down /= k;

      cout << up;

      down == 1 ? cout << "" : cout << "/" << down;

   }; // display rational number

   private:

   int up; // private variable numerator

   int down; // private variable denominator

   void reduction(); // function for fraction reduction

}; // end class RationalNumber

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

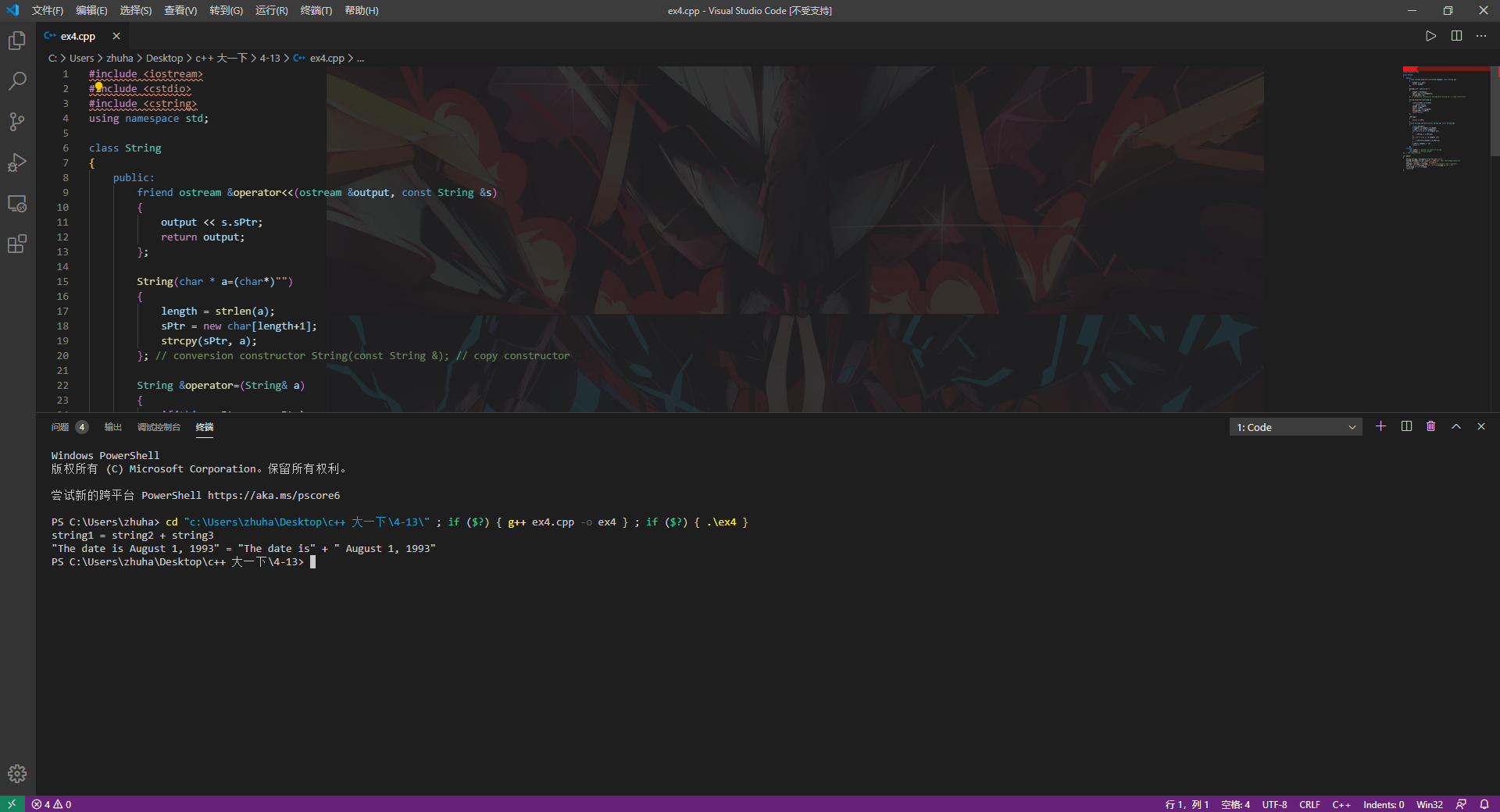
    cout << ( ( c != d ) ? "  != " : "  == " );

    d.printRational();

    cout << " according to the overloaded != operator" << endl; return 0;

    }

Ex4



#include <iostream>

#include <cstdio>

#include <cstring>

using namespace std;

class String

{

    public:

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

        {

            output << s.sPtr;

            return output;

        };

        String(char \* a=(char\*)"")

        {

            length = strlen(a);

            sPtr = new char[length+1];

            strcpy(sPtr, a);

        }; // conversion constructor String(const String &); // copy constructor

        String &operator=(String& a)

        {

            if(this->sPtr == a.sPtr)

                return \*this;

            length = a.length;

            delete []sPtr;

            sPtr = new char[length];

            strcpy(sPtr, a.sPtr);

            return \*this;

        };

        ~String()

        {

            delete [] sPtr;

        }

        friend String& operator+(const String &a, const String &b)

        {

            static String r;

            r.length = a.length + b.length;

            r.sPtr = new char[r.length+1];

            for (int j = 0; j < a.length; j++)

            {

                r.sPtr[j] = a.sPtr[j];

            }

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

            {

                r.sPtr[i+a.length] = b.sPtr[i];

            }

            r.sPtr[r.length] = '\0';

            return r;

        };

    private:

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

        int length; // string length

}; // end class String

int main()

{

    String string1, string2((char\*)"The date is");

    String string3((char\*)" August 1, 1993");// test overloaded operators

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

    string1 = string2 + string3; // tests overloaded = and + operator

    cout << "\"" << string1 << "\" = \"" << string2 << "\" + \""

    << string3 << "\"" << endl;

    return 0;

}