```
#include <string>
#include <iostream>
#include <cmath>
#include <sstream>

using namespace std;

class ComplexNumbers
{
private:
[TAB]double real, imag;

public:
[TAB]static const double DEFAULT_NUM;
[TAB]ComplexNumbers(double realNum = DEFAULT_NUM, double imagNum = DEFAULT_NUM);
[TAB]ComplexNumbers reciprocal();
[TAB]ComplexNumbers& operator =(ComplexNumbers& comNum1);
[TAB]double getReal() const { return real; }
[TAB]double getImag() const { return imag; }

[TAB]bool setReal(double newReal);
[TAB]bool setImag(double newImag);

[TAB]double modulus() const;

[TAB]string toString() const;

[TAB]// operators
[TAB]friend ComplexNumbers operator +(const ComplexNumbers& comNum1,
[TAB][TAB]const ComplexNumbers& comNum2);
[TAB]friend ComplexNumbers operator -(const ComplexNumbers& comNum1,
[TAB][TAB]const ComplexNumbers& comNum2);
[TAB]friend ComplexNumbers operator *(const ComplexNumbers& comNum1,
[TAB][TAB]const ComplexNumbers& comNum2);
[TAB]friend ComplexNumbers operator /(const ComplexNumbers& comNum1,
[TAB][TAB]const ComplexNumbers& comNum2);
[TAB]friend bool operator <(const ComplexNumbers& comNum1,
[TAB][TAB]const  ComplexNumbers& comNum2);
[TAB]friend bool operator ==(const ComplexNumbers& comNum1,
[TAB][TAB]const ComplexNumbers& comNum2);
[TAB]friend ostream& operator << (ostream& outputStream,
[TAB][TAB]const ComplexNumbers& comNum);

 [TAB]class DivByZero {};
};
const double ComplexNumbers::DEFAULT_NUM = 0.0;

int main()
{
[TAB]try
[TAB]{
[TAB][TAB]ComplexNumbers a(3, -4), b(1.1, 2.1), c;
[TAB][TAB]double x = 2, y = -1.7;
[TAB][TAB]cout << "a is " << a << "b is " << b
[TAB][TAB][TAB]<< "x is " << x << endl << "y is " << y << endl;
[TAB][TAB]c = a + b;
[TAB][TAB]cout << "a + b = " << c;
[TAB][TAB]c = x - a;
[TAB][TAB]cout <<"x - a = " << c;
[TAB][TAB]c = b * y;
[TAB][TAB]cout <<"a * y = " << c;
[TAB][TAB]// and also:
```

```
[TAB][TAB]c = 8 + a;
[TAB][TAB]cout <<"8 + a = " << c;
[TAB][TAB]c = b / 3.2;
[TAB][TAB]cout << "b / 3.2 = " << c;

[TAB][TAB]if (a == b)
[TAB][TAB][TAB]cout << "a and b are the same." << endl;
[TAB][TAB]else
[TAB][TAB]{
[TAB][TAB][TAB]cout << "a and b are not the same." << endl;
[TAB][TAB]}
[TAB][TAB]if (a < b)
[TAB][TAB][TAB]cout << "a is less the b." << endl;
[TAB][TAB]else
[TAB][TAB]{
[TAB][TAB][TAB]cout << "a is greater then b." << endl;
[TAB][TAB]}
[TAB][TAB]c = b / 0;
[TAB][TAB]cout << c;
[TAB]}
[TAB]
[TAB]catch (ComplexNumbers::DivByZero)
[TAB]{
[TAB][TAB]cout << "Can't divid by 0" << endl;
[TAB]}
}
ComplexNumbers::ComplexNumbers(double realNum, double imagNum) :
[TAB]real(realNum), imag(imagNum){}

ComplexNumbers ComplexNumbers::reciprocal()
{
[TAB]if (modulus() < 0.00001)
[TAB][TAB]throw DivByZero();

[TAB]// complex number = ( r / (r*r + i*i), -i / (r*r + i*i) ), if (r*r +
[TAB]//i*i) is not zero.If(r*r + i*i) is zero, then reciprocal() throws an excep
tion.
[TAB]double newReal = (real / ((real * real) + (imag * imag)));
[TAB]double newImag = (-imag / ((real * real) + (imag * imag)));

[TAB]return ComplexNumbers(newReal, newImag);
}
bool ComplexNumbers::setImag(double newImag)
{
[TAB]if (imag = newImag)
[TAB][TAB]return true;
[TAB]else
[TAB][TAB]return false;
}
bool ComplexNumbers::setReal(double newReal)
{
[TAB]if (real = newReal)
[TAB][TAB]return true;
[TAB]else
[TAB][TAB]return false;
}
double ComplexNumbers::modulus() const
{
[TAB]double modulus;
[TAB]
[TAB]modulus = sqrt((real * real) + (imag * imag));
[TAB]return modulus;
```

```
}
string ComplexNumbers::toString() const
{
[TAB]string results;
[TAB]ostringstream numStream;
[TAB]
[TAB]numStream << "( " << real << ", " << imag << " )";
[TAB]results = numStream.str();
[TAB]return results;
}
ComplexNumbers operator +(const ComplexNumbers& comNum1,
[TAB]const  ComplexNumbers& comNum2)
{
[TAB]double realNumbers1 = comNum1.real + comNum2.real;
[TAB]double Imaginary = comNum1.imag + comNum2.imag;
[TAB]
[TAB]return ComplexNumbers(realNumbers1, Imaginary);
}
ComplexNumbers operator -(const ComplexNumbers& comNum1,
[TAB]const  ComplexNumbers& comNum2)
{
[TAB]double realNumbers1 = comNum1.real - comNum2.real;
[TAB]double Imaginary = comNum1.imag - comNum2.imag;

[TAB]return ComplexNumbers(realNumbers1, Imaginary);
}
ComplexNumbers operator *(const ComplexNumbers& comNum1,
[TAB]const  ComplexNumbers& comNum2)
{
[TAB]//(r,i) * (s,j) = (r*s - i*j, r*j + s*i).
[TAB]double realNumbers1 = comNum1.real * comNum2.real;
[TAB]double Imaginary = comNum1.imag * comNum2.imag;
[TAB]double realImag1 = comNum1.real * comNum2.imag;
[TAB]double realImag2 = comNum1.imag * comNum2.real;
[TAB]double subSum = realNumbers1 - Imaginary;
[TAB]double addSum = realImag1 + realImag2;

[TAB]return ComplexNumbers(subSum, addSum);
}
ComplexNumbers operator /(const ComplexNumbers& comNum1,
[TAB]const ComplexNumbers& comNum2)
{
[TAB]ComplexNumbers number2 = comNum2;
[TAB]ComplexNumbers number1 = comNum1 * number2.reciprocal();

[TAB]return number1;
}
bool operator ==(const ComplexNumbers& comNum1, const ComplexNumbers& comNum2)
{
[TAB]return ((comNum1.real == comNum2.real)
[TAB][TAB]&& (comNum1.imag == comNum2.imag));
}
bool operator <(const ComplexNumbers& comNum1, const ComplexNumbers& comNum2)
{
[TAB]return (comNum1.modulus() < comNum2.modulus());
}
ComplexNumbers& ComplexNumbers::operator =(ComplexNumbers& comNum1)
{
[TAB]setReal(comNum1.real);
[TAB]setImag(comNum1.imag);
[TAB]return ComplexNumbers(comNum1);
}
```

```
ostream& operator <<(ostream& outputStream, const ComplexNumbers& comNum)
{
[TAB]outputStream << comNum.toString() << endl;
[TAB]return outputStream;
}

/*------------------------------ Posted Run ----------------------------------
-
a is ( 3, -4 )
b is ( 1.1, 2.1 )
x is 2
y is -1.7
a + b = ( 4.1, -1.9 )
x - a = ( -1, 4 )
a * y = ( -1.87, -3.57 )
8 + a = ( 11, -4 )
b / 3.2 = ( 0.34375, 0.65625 )
a and b are not the same.
a is greater then b.
Can't divid by 0
Press any key to continue . . .
--------------------------------------------------------------------------------
-*/
```

```
// CS 2B Lab 5
// Instructor Solution:
//  Original - Prof. Loceff, Updates, Edits, Annotations: &

// Notes:
//  - Use of sensible names for vars
//  - Correct arithmetic
//  - Faithfulness to spec (reporting per serving, etc.)
//  - Correct definition and usage of Exception
//  - Correct method qualifications
//  - Correct handling of round-off error in reciprocal()
//
#include <iostream>
#include <sstream>
#include <string>
#include <cmath>
#include <stack>

using namespace std;

// Complex prototype ----------------------------------
class Complex {
    // friend operators
    friend Complex operator+(const Complex& a, const Complex& b);
    friend Complex operator-(const Complex& a, const Complex& b);
    friend Complex operator*(const Complex& a, const Complex& b);
    friend Complex operator/(const Complex& a, const Complex& b);
    friend bool operator==(const Complex& a, const Complex& b);
    friend bool operator<(const Complex& a, const Complex& b);
    friend ostream& operator<<(ostream& os, const Complex& x);

private:
    double real;
    double imag;

public:
    Complex (double re = 0.0, double im = 0.0) : real(re), imag(im) {};

    void setReal(double re) { real = re; }
    void setImag(double im) { imag = im; }

    double getReal() const { return real; }
    double getImag() const { return imag; }
    string toString() const;

    double norm() const { return real*real + imag*imag; }
    double modulus() const { return sqrt(norm()); }
    const Complex reciprocal() const;

    const Complex& operator= (const Complex & rhs);

    class DivByZeroException {
    public:
```

```
        string toString() { return "Zero Denominator Exception"; }
        string what() { return toString(); } // more conventional
    };
};

// ----- Out of line Complex method defs -----

const Complex Complex::reciprocal() const {
    double theNorm = norm();

    if (theNorm <= 1e-10) // watch for round-off
        throw DivByZeroException();

    return Complex(real/theNorm , -imag/theNorm);
}

// Complex::toString -----------------------------
string Complex::toString() const {
    ostringstream os;
    os << "(" << real << ", " << imag << ")";
    return os.str();
}

const Complex& Complex::operator=(const Complex& rhs) {
    if (this == &rhs)
        return *this;
    real = rhs.real;
    imag = rhs.imag;
    return *this;
}

// Note: ALL the following can be used for either Complex OR double due to
// implied constructor call
Complex operator+(const Complex& a, const Complex& b) {
    return Complex(a.real + b.real, a.imag + b.imag);
}
Complex operator-(const Complex& a, const Complex& b) {
    return Complex(a.real - b.real, a.imag - b.imag);
}
Complex operator*(const Complex& a, const Complex& b) {
    return Complex(a.real * b.real - a.imag * b.imag,
                   a.real * b.imag + b.real * a.imag);
}
Complex operator/(const Complex& a, const Complex& b) {
    return a * b.reciprocal();
}

bool operator==(const Complex& a, const Complex& b) {
    return ((a.real == b.real) && (a.imag == b.imag));
}
bool operator<(const Complex& a, const Complex& b) {
    return (a.modulus() < b.modulus());
}
```

```
ostream& operator<<(ostream &out, const Complex& x) {
    out << x.toString();
    return out;
}

// ----- Test Driver -----
int main () {
    // -- Option A --
    Complex a(1,2), b(3,4), c;

    cout << a << " + " << b <<  " = ";
    c = a + b;
    cout << c << endl;

    cout << a << " - " << b <<  " = ";
    c = a - b;
    cout << c << endl;

    cout << a << " * " << b <<  " = ";
    c = a * b;
    cout << c << endl;

    try {
        cout << a << " / " << b <<  " = ";
        c = a / b;
        cout << c << endl;
    } catch (Complex::DivByZeroException e) {
        cout << e.what() <<endl;
    }

    try {
        cout << a << " / " << 0 <<  " = ";
        c = a / 0;
        cout << c << endl;
    } catch (Complex::DivByZeroException e) {
        cout << e.what() <<endl;
    }

    cout << a << " + 10 = ";
    c = a + 10;
    cout << c << endl;

    cout << "10 / " << b <<  " = ";
    c = 10 / b;
    cout << c << endl;

    // -- Option B --
    int total;
    string userResp;
    stack<Complex> cStack;

    cout << "How many complex numbers should I generate? ";
```

```
    getline(cin, userResp);
    istringstream(userResp) >>total;

    srand((unsigned) time(0L));
    for (int k = 0; k < total ; k++) {
        c.setImag(rand());
        c.setReal(rand());
        cout << "pushing " << c << endl;
        cStack.push(c);
    }

    cStack.push(Complex(3,5));
    cStack.push(Complex(1,1) + Complex(2,4));

    while (cStack.size() >= 2) {
        a = cStack.top(); cStack.pop();
        b = cStack.top(); cStack.pop();

        cout << a.modulus() << " vs. " << b.modulus() << endl;
        if (a < b)
            cout << a << " < " << b << endl;
        else if (b < a)
            cout << a << " > " << b << endl;
        else
            cout << "|" << a << "| = |" << b << "|" << endl;

    }

    return 0;
}

/* -- Run --
 (1, 2) + (3, 4) = (4, 6)
 (1, 2) - (3, 4) = (-2, -2)
 (1, 2) * (3, 4) = (-5, 10)
 (1, 2) / (3, 4) = (0.44, 0.08)
 (1, 2) / 0 = Zero Denominator Exception
 (1, 2) + 10 = (11, 2)
 10 / (3, 4) = (1.2, -1.6)
 How many complex numbers should I generate? 5
 pushing (7.81978e+08, 1.90681e+09)
 pushing (6.98553e+08, 9.90658e+07)
 pushing (6.52285e+08, 2.89828e+08)
 pushing (1.55139e+08, 4.49854e+07)
 pushing (1.77378e+08, 3.71958e+08)
 5.83095 vs. 5.83095
 |(3, 5)| = |(3, 5)|
 4.12087e+08 vs. 1.61529e+08
 (1.77378e+08, 3.71958e+08) > (1.55139e+08, 4.49854e+07)
 7.13776e+08 vs. 7.05543e+08
 (6.52285e+08, 2.89828e+08) > (6.98553e+08, 9.90658e+07)
 Program ended with exit code: 0
 */
```