**CSCI 241 Assignment 4  
100 points**

This assignment introduces the concept of *operator overloading*, the use of the friend keyword, and the concept of "const correctness".

**Assignment**

A number of the form *a* + *ib*, where *i*2 = -1 and *a* and *b* are real numbers, is called a **complex number**. We call *a* the **real part** and *b* the **imaginary part** of *a* + *ib*. Complex numbers can also be represented as ordered pairs (*a*, *b*). The addition and multiplication of complex numbers is defined by the following rules:

(*a* + *ib*) + (*c* + *id*) = (*a* + *c*) + *i*(*b* + *d*)

(*a* + *ib*) \* (*c* + *id*) = (*ac* - *bd*) + *i*(*ad* + *bc*)

Using the ordered pair notation, these rules are written as:

(*a*, *b*) + (*c*, *d*) = ((*a* + *c*), (*b* + *d*))

(*a*, *b*) \* (*c*, *d*) = ((*ac* - *bd*), (*ad* + *bc*))

C++ has no built-in data type that allows us to manipulate complex numbers. In this assignment, we will construct a new data type, Complex, that can be used to process complex numbers. We will overload the stream insertion and stream extraction operators for easy input and output. We will also overload the operators + and \* to perform addition and multiplication of complex numbers. If *x* and *y* are complex numbers, we can evaluate expressions such as *x* + *y* and *x* \* *y*.

More information on complex numbers can be found [here](http://en.wikipedia.org/wiki/Complex_number), but the description given above should be sufficient to complete this assignment.

**Program**

You will need to write one class for this assignment. A main program to test your class has been provided.

**The Complex class**

The Complex class represents a complex number as an ordered pair. Like the other classes we've written this semester, this class should be implemented as two separate files.

The class definition should be placed in a header file called Complex.h. Include header guards to prevent it from accidentally being #included more than once in the same source code file.

The Complex class should contain the following private data members:

* a double to store the real part of the complex number.
* another double to store the imaginary part of the complex number.

In addition to the data members described above, the class definition should also contain prototypes for the methods described below.

The implementations of the class methods should be placed in a separate source code file called Complex.cpp. Make sure to #include "Complex.h" at the top of this file.

The Complex class should have the following methods (most of which are quite small):

* Complex constructor

*Parameters:* The constructor should take two double variables as arguments, representing the real and imaginary parts of a complex number. Give the two parameters the default value 0 in the prototype for the constructor (which will allow this constructor to double as a "default" constructor).

*Logic:* Assign the method parameters to the corresponding data members.

* setComplex()

*Parameters:* This method takes two double variables as arguments, representing the real and imaginary parts of a complex number.

*Returns:* Nothing.

*Logic:* Assign the method parameters to the corresponding data members.

* getComplex()

*Parameters:* This method takes two references to double variables as arguments, representing the real and imaginary parts of a complex number.

*Returns:* Nothing.

*Logic:* Assign the data members to the corresponding method parameters. This lets us pass back the value of both parts of the complex number with a single method call.

This method does not alter any data members of the object that called the method, so it should be declared const.

* setRealPart()

*Parameters:* A double, representing the real part of a complex number.

*Returns:* Nothing.

*Logic:* Assign the method parameter to the real part data member of the object that called the method.

* getRealPart()

*Parameters:* None.

*Returns:* The real part of the complex number (a double).

*Logic:* Return the data member representing the real part of the complex number.

This method does not alter any data members of the object that called the method, so it should be declared const.

* setImaginaryPart()

*Parameters:* A double, representing the imaginary part of a complex number.

*Returns:* Nothing.

*Logic:* Assign the method parameter to the imaginary part data member of the object that called the method.

* getImaginaryPart()

*Parameters:* None.

*Returns:* The imaginary part of the complex number (a double).

*Logic:* Return the data member representing the imaginary part of the complex number.

This method does not alter any data members of the object that called the method, so it should be declared const.

* operator+()

This method will be called when the operator + is used to add two Complex objects. For example:

// Assume c1, c2, and c3 are Complex objects

c3 = c1 + c2 // Generates this method call: c1.operator+(c2);

*Parameters:* This method takes one parameter, a reference to a constant Complex object, representing the right operand of the arithmetic expression. The left operand of the expression is represented bythis, which points to the Complex object that called the method.

*Returns:* A Complex object that holds the result of the arithmetic.

*Logic:* Declare a Complex object to hold the result of the arithmetic. You will need to set the real part and the imaginary part of this result object to the correct values, as outlined in the ordered pair version of the addition rule shown at the beginning of the assignment sheet. For example, the real part of the result object should be set to the sum of the real part of the left operand (which corresponds to the variable *a* in the rule) and the real part of the right operand (which corresponds to the variable *c* in the rule). Once you've done the arithmetic, return the result object.

This method does not alter any data members of the object that called the method, so it should be declared const.

* operator\*()

This method will be called when the operator \* is used to multiply two Complex objects. For example:

// Assume c1, c2, and c3 are Complex objects

c3 = c1 \* c2 // Generates this method call: c1.operator\*(c2);

*Parameters:* This method takes one parameter, a reference to a constant Complex object, representing the right operand of the arithmetic expression. The left operand of the expression is represented bythis, which points to the Complex object that called the method.

*Returns:* A Complex object that holds the result of the arithmetic.

*Logic:* Declare a Complex object to hold the result of the arithmetic. You will need to set the real part and the imaginary part of this result object to the correct values, as outlined in the ordered pair version of the multiplication rule shown at the beginning of the assignment sheet. Once you've done the arithmetic, return the result object.

This method does not alter any data members of the object that called the method, so it should be declared const.

* operator==()

This method will be called when the operator == is used to compare two Complex objects. For example:

// Assume c1 and c2 are Complex objects

if (c1 == c2) // Generates this method call: c1.operator==(c2);

*Parameters:* This method takes one parameter, a reference to a constant Complex object, representing the right operand of the relational expression. The left operand of the expression is represented bythis, which points to the Complex object that called the method.

*Returns:* A boolean value.

*Logic:* The method should return true if the real part of the left operand equals the real part of the right operand AND the imaginary part of the left operand equals the imaginary part of the right operand. Otherwise, the method should return false.

This method does not alter any data members of the object that called the method, so it should be declared const.

In addition to the methods described above, you will need to write two standalone functions. These functions are not (and can not be) methods. You should

1. Include a friend declaration for each of these functions in the Complex class definition.
2. Put the definitions for these functions in Complex.cpp.

* operator<<()

This function will be called when the stream insertion operator << is used to print a Complex object. For example:

Complex c(3, 5); // A complex number

cout << c; // Generates this function call: operator<<(cout, c);

In the example code above, the ostream object cout (the left operand in the stream insertion expression) will be passed to the function as the first parameter, while the Complex object c (the right operand) will be passed to the function as the second parameter.

*Parameters:* This function takes two parameters. The first is a reference to an ostream object, representing the left operand of the stream insertion expression. The second is a reference to a constant Complex object, representing the right operand of the expression.

*Returns:* A reference to an ostream object (i.e., the first parameter).

*Logic:* This function should print a Complex object as an ordered pair of the form:

(*real part*, *imaginary part*)

So printing the object c from the example above should produce the output:

(3, 5)

You can use the left operand parameter to print a left parenthesis:

lhs << '('; // lhs is the reference to an ostream object

// passed into the method as the 1st parameter

In similar fashion, you can use the left operand and the << operator to print the real part of the right operand, followed by a comma, a space, the imaginary part of the right operand, and a right parenthesis. At the end of the function, return the left operand (to allow cascading of the << operator to work correctly).

* operator>>()

This function will be called when the stream extraction operator >> is used to read a Complex object. For example:

Complex c; // A complex number

cin >> c; // Compiler generated function call: operator>>(cin, c);

In the example code above, the istream object cin (the left operand in the stream extraction expression) will be passed to the function as the first parameter, while the Complex object c (the right operand) will be passed to the function as the second parameter.

*Parameters:* This function takes two parameters. The first is a reference to an istream object, representing the left operand of the stream extraction expression. The second is a reference to a Complexobject, representing the right operand of the expression.

*Returns:* A reference to an istream object (i.e., the first parameter).

*Logic:* This function should read input entered by the user as an ordered pair of the form:

(*real part*, *imaginary part*)

You can use the left operand parameter to read the left parenthesis entered by the user into a char variable:

char ch;

lhs >> ch; // lhs the reference to an istream object

// passed into the method as the 1st parameter

(You don't need to do anything with parenthesis once it's been read. We just need to get it out of the way so we can read the stuff that's important.)

Once the left parenthesis has been read, you can then read the real part of the input entered by the user into the real part of the right operand object. Then read the comma into your char variable. Read the imaginary part of the input entered by the user into the imaginary part of the right operand object. Then read the right parenthesis entered by the user into the char variable. Finally, return the left operand (to allow cascading of the << operator to work correctly).

**Driver Program**

A short main program to test your class is given below. Either type in (or copy and paste) this program or copy it to your UNIX account from the pathname ~t90kjm1/CS241/Code/Fall2017/Assign4/assign4.cpp.

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

PROGRAM: CSCI 241 Assignment 4

PROGRAMMER: your name

LOGON ID: your z-ID

DUE DATE: due date of assignment

FUNCTION: This program tests the functionality of the Complex

class.

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

#include <iostream>

#include <iomanip>

#include "Complex.h"

using std::cin;

using std::cout;

using std::endl;

int main()

{

cout << "Testing constructor... ";

Complex c1(23, 34);

const Complex c4(3, 4);

cout << "done\n";

cout << "Testing default constructor use... ";

Complex c2;

Complex c3;

cout << "done\n\n";

cout << "Testing stream insertion operator and constructors...\n";

cout << "c1 = " << c1 << endl;

cout << "c2 = " << c2 << endl << endl;

cout << "Testing get methods...\n";

cout << "Real part of c4 = " << c4.getRealPart() << endl;

cout << "Imaginary part of c4 = " << c4.getImaginaryPart() << endl;

double r, i;

c4.getComplex(r, i);

cout << "Real part of c4 = " << r << endl;

cout << "Imaginary part of c4 = " << i << endl << endl;

cout << "Testing set methods...\n";

c2.setComplex(3.7, 2.5);

cout << "New value of c2 = " << c2 << endl;

c2.setRealPart(-1.4);

cout << "New value of c2 = " << c2 << endl;

c2.setImaginaryPart(83);

cout << "New value of c2 = " << c2 << endl << endl;

cout << "Testing stream extraction operator...\n";

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

cin >> c2;

cout << "New value of c2 = " << c2 << endl << endl;

cout << "Testing addition operator...\n";

c3 = c1 + c4;

cout << "c3 = " << c3 << endl;

cout << c4 << " + " << c1 << " = " << c4 + c1 << endl;

cout << c4 << " + " << c4 << " = " << c4 + c4 << endl << endl;

cout << "Testing multiplication operator...\n";

c3 = c1 \* c4;

cout << "c3 = " << c3 << endl;

cout << c4 << " \* " << c1 << " = " << c4 \* c1 << endl;

cout << c4 << " \* " << c4 << " = " << c4 \* c4 << endl << endl;

cout << "Testing equality operator...\n";

cout << c1 << " and " << c3;

(c1 == c3) ? cout << " are equal\n" : cout << " are not equal\n";

const Complex c5(3, 4);

cout << c4 << " and " << c5;

(c4 == c5) ? cout << " are equal\n" : cout << " are not equal\n";

return 0;

}

**Sample Output**

A successful run of the entire program should produce output that resembles the output below. Note that in response to the prompt "Enter a complex number in the form (a, b) ", the user entered "(2, 7)".

Testing constructor... done

Testing default constructor... done

Testing stream insertion operator and constructors...

c1 = (23, 34)

c2 = (0, 0)

Testing get methods...

Real part of c4 = 3

Imaginary part of c4 = 4

Real part of c4 = 3

Imaginary part of c4 = 4

Testing set methods...

New value of c2 = (3.7, 2.5)

New value of c2 = (-1.4, 2.5)

New value of c2 = (-1.4, 83)

Testing stream extraction operator...

Enter a complex number in the form (a, b) (2, 7)

New value of c2 = (2, 7)

Testing addition operator...

c3 = (26, 38)

(3, 4) + (23, 34) = (26, 38)

(3, 4) + (3, 4) = (6, 8)

Testing multiplication operator...

c3 = (-67, 194)

(3, 4) \* (23, 34) = (-67, 194)

(3, 4) \* (3, 4) = (-7, 24)

Testing equality operator...

(23, 34) and (-67, 194) are not equal

(3, 4) and (3, 4) are equal

**Other Points**

* A makefile is required. Use the makefile for Assignment 1 as your model.
* Many of the overloaded operators for this assignment can be coded either as methods of the Complex class **or** as standalone code functions. The descriptions in this assignment assume that you will code the overloaded operators as methods whenever possible, but you are welcome to code them as functions if your prefer.
* The driver program is designed to make this assignment easy to develop incrementally. Simply comment out all of the lines of the driver program that call methods or functions you haven't written yet. You should be able to write, test, and debug one or two methods or functions at a time:
  1. constructor
  2. operator<<()
  3. getRealPart(), getImaginaryPart(), getComplex()
  4. setRealPart(), setImaginaryPart(), setComplex()
  5. operator>>()
  6. operator+()
  7. operator\*()
  8. operator==()
* As always, programs that do not compile on turing/hopper automatically receive 0 points.
* Make sure to document your program according to the standards listed in the Course Notes.
* Submit your program using the electronic submission guidelines posted on the course web site and described in class.