CptS 122 – Data Structures

Lab 6: The Wonderful World of C++, Classes, and Objects

Assigned: Monday, June 26, 2017
Due: At the end of the lab session

I. Learner Objectives:

At the conclusion of this programming assignment, participants should be able to:

- Design, implement and test classes in C++
- Declare and define *constructors*
- Declare and define *destructors*
- Compare and contrast *public* and *private* access specifiers in C++
- Describe what is an *attribute* or data member of a class
- Describe what is a *method* of a class
- Apply and implement *overloaded* functions
- Distinguish between pass-by-*value* and pass-by-*reference*
- Discuss *classes* versus *objects*

II. Prerequisites:

Before starting this programming assignment, participants should be able to:

- Analyze a basic set of requirements for a problem
- Compose a small C++ language program
- Create test cases for a program

III. Overview & Requirements:

This lab will allow you to further explore lists, and navigate through designing, implementing, and testing classes in C++.

Labs are held in a "closed" environment such that you may ask your TA questions. Please use your TA's knowledge to your advantage. You are required to move at the pace set forth by your TA. Please help other students in need when you are finished with a task. Have a great time! Labs are a vital part to your education in CptS 122 so work diligently.

Tasks:

1. In teams, before you implement any code, discuss, with the use of the whiteboard, a general C++ class that will satisfy the requirements for the application described below. Complete programming project Complex numbers from your Deitel and Deitel C++ How to Program book, with some additional tasks provided by me.

Create a class called *Complex* for performing arithmetic with complex numbers. Write a program to test your class. Complex numbers have the form:

realPart + imaginaryPart * i

where *i* is

sqrt (-1)

use double variables to represent the private data of the class. Provide a constructor that enables an object of this class to be initialized when it is declared. The constructor should contain default values in case no initializers are provided. Define *public* setters and getters to access the private data members. Below in Figure 1 is a summary of the Complex Number ADT:

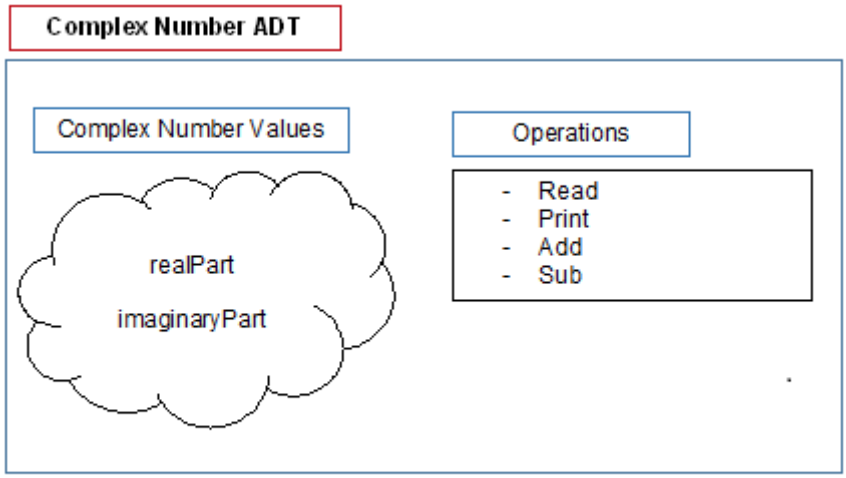

Figure 1: Complex Number ADT

Also, provide functions that perform the following tasks:

a) Adding two *Complex* numbers: The real parts are added together and the imaginary parts are added together. Implement the addition operation in three ways:

i. Implement a *member* function of class *Complex* called `add()` that must do the following: accept one *Complex* number *rhs* for an argument, add *rhs* to the data members in the object that invokes the `add()` function, and return the result.

ii. Implement a *non-member* function called `add()`. Make sure it has the same name as the one defined in part (i). Place the prototype/declaration for this function outside of the *Complex* number class's declaration (below the class declaration). Place its definition inside of the Complex.cpp file. The function must do the following: accept two *Complex* numbers called *lhs* and *rhs* for arguments, add *lhs* and *rhs* together, and return the result.

iii. Implement a *non-member* overloaded addition (+) *operator*. Place the prototype/declaration for this function outside of the *Complex* number class's declaration (below the class declaration). Place its definition inside of the Complex.cpp file. The function must do the following: accept two *Complex* numbers called *lhs* and *rhs* for arguments, add *lhs* and *rhs* together, and return the result. Note: the overloaded + is a binary operation, which requires two arguments! Use the following prototype/declaration:

```
Complex operator+ (const Complex &lhs, const Complex &rhs);
```

Note: overloading the + operator allows for us to use statements such as: c3 = c1 + c2, where c1, c2, and c3 are *Complex* numbers. Another interpretation is: c3 = operator+(c1, c2).

Now place the following statements in a test driver, and use the debugger to watch each of the *Complex* numbers:

```
Complex c1(3.5, 2), c2(5.5, 7), c3;

c3 = c1.add(c2);  // member add () function
c3 = add(c1, c2); // non-member add () function
c3 = c1 + c2;     // overloaded + operator
```

b) Subtracting two *Complex* numbers: The real part of the right operand is subtracted from the real part of the left operand, and the imaginary part of the operand. Solve this problem by overloading the subtraction (-) *operator*. Use the debugger to watch each of the *Complex* numbers declared for part (a).

```
c3 = c1 - c2;     // overloaded - operator
```

c) Reading *Complex* numbers from the keyboard, in the form a + bi, where *a* is the real part and *b* is the imaginary part. Implement the read operation in two ways:

i. Implement a *member* function of class *Complex* called `read()` that must do the following: accept no arguments, read in the real and imaginary parts of the number from the standard *input* stream in the form:

a + bi,

and return nothing. Yes, you must read in the + (or minus -) and the i, but they should be discarded. This function does not prompt for the *Complex* number. The prompt is done external to the function.

ii. Implement a *non-member* overloaded stream extraction (>>) *operator*. Place the prototype/declaration for this function outside of the *Complex* number class's declaration (below the class declaration). Place its definition inside of the Complex.cpp file. The function must do the following: accept one *istream* object called *lhs* and one *Complex* number called *rhs* for arguments, extract the real and imaginary parts of the number from the standard *input* stream in the form:

a + bi,

and return the i*stream* object (so we can chain >> together!). Note: once again, you should discard the the + (or minus -) and the i. Use the following prototype/declaration:

```
istream & operator>> (istream &lhs, Complex &rhs);
```

Assuming that you still have the same Complex numbers instantiated: c1, c2, and c3. Use the debugger to watch each of the *Complex* numbers declared. Also, place the following statements in a test driver:

```
cout << "Enter a complex number in the form a + bi: ";
c1.read();
cout << "Enter a complex number in the form a + bi: ";
c2.read();

cout << "Enter two complex numbers in the form a + bi (each separated by whitespace): ";
```

```
cin >> c1 >> c2;
```

d) Printing *Complex* numbers to the screen, in the form a + bi, where *a* is the real part and *b* is the imaginary part. Implement the print operation in two ways:

j. Implement a *member* function of class *Complex* called `print()` that must do the following: accept no arguments, insert the real and imaginary parts of the number into the standard *output* stream in the form:

a + bi,

and return nothing.

jj. Implement a *non-member* overloaded stream insertion (<<) *operator*. Place the prototype/declaration for this function outside of the *Complex* number class's declaration (below the class declaration). Place its definition inside of the Complex.cpp file. The function must do the following: accept one *ostream* object called *lhs* and one *Complex* number called *rhs* for arguments, insert the real and imaginary parts of the number into the standard *output* stream in the form:

a + bi,

and return the *ostream* object (so we can chain << together!). Use the following prototype/declaration:

```
ostream & operator<< (ostream &lhs, const Complex &rhs);
```

Once again, assuming that you still have the same Complex numbers instantiated: c1, c2, and c3. Place the following statements in a test driver:

```
c1.print();
c2.print();
c3.print();

cout << c1 << " " << c2 << " " << c3 << endl;
```

2. Read the following http://www.consumerfinance.gov/askcfpb/309/what-is-a-credit-report.html. Once again work with your team to write and test a class called CreditReport. This class represents a real-world credit report. In this problem we will model the credit report in the following way. It should contain a credit score (ranges 330 – 830), a debt profile (total real estate and credit card debt), account types (number of real estate, credit cards, and retail cards), length of history (oldest account age and average account age), and number of hard inquiries (number of times your credit report has been accessed) as attributes. Operations that may be applied to your CreditReport include: printReport and updateReport (you may refine updateReport for use with individual class attributes). Be sure to include all necessary getters and setters, constructors, and destructors. Be sure to define a copy constructor! Write an application which instantiates three credit reports. These include Experian, TransUnion, and Equifax. Your application should decrease credit scores as the credit limit on cards is approached and increase scores as the credit is paid off. Also the older the credit accounts, the higher the credit score. Also, use the other attributes of the CreditReport as you see fit.

3. Work on assignment 4.

IV. Submitting Labs:

- You are not required to submit your lab solutions. However, you should keep them in a folder that you may continue to access throughout the semester. You should not store your solutions to the local C: drive on the EME 120/128 machines. These files are erased on a daily basis.

V. Grading Guidelines:

- This lab is worth 10 points. Your lab grade is assigned based on completeness and effort. To receive full credit for the lab you must show up on time and continue to work on the problems until the TA has dismissed you.