ECE2036 **Spring Semester, 2016**

Project 3 – Complex Number Calculator

Assigned: Feb 10, 2016 *Due: Feb 24, 2016*

In this assignment, we will create a calculator that performs simple arithmetic operations on complex numbers. Complex values are denoted by a parenthesized pair of values separated by a comma representing the real and imaginary part of the variable. For example `(1,2)` indicates that the real part is 1 and the imaginary part is 2. A complex number can also be represented by the magnitude and angle format like this `(1 > 45)` indicating a complex value with a magnitude of 1 and an angle of 45 degrees. Finally, a single numeric value without parenthesis indicates a complex number with the real value equal to the specified numeric value and an imaginary part of zero.

There are three basic requirements for the calculator.

1. If the input line has two operands and one operator, the calculator should perform the specified operation and print the result. Further, the result should be saved for use in requirement 2 below. An example is:

```
(1,2) + (2,3)          (input by user)
    = (3,5)            (output by the program)
```

2. If the input line starts with an operator followed by a single operand, the result from the prior operation is used as the left–side operand. An example is:

```
(1,2) + (2,3)          (input by user)
    = (3,5)            (output by the program)
+ (5,5)                (input by user)
    = (8,10)           (output by the program)
```

3. Finally, a single operand with no operator simply outputs the value of the single operand and stores it as the prior result.

```
(1,2)                  (input by user)
    = (1,2)            (output by the program)
+ (5,5)                (input by user)
    = (6,7)            (output by the program)
```

In order to complete this assignment, your program must be written in C++ and implement the functions described in the skeleton code provided. You will need to implement the `Complex` class, and provide overloaded operators for the plus, minus, multiply, and divide operators. You will *NOT* need an exponentiation operator for this assignment. For the addition and subtraction operators use *Member Function Operator Overloads* and for the multiplication and division use the *Non-Member Function Operator Overloads*. We will discuss in class the difference in these two approaches.

The `Complex` class will need a constructor with no arguments (default constructor), one with two arguments with initial values of both the real and imaginary part, and a third constructor that builds a complex number from a `const string&`. We have not discussed the `string` class yet, but it is simply a class that manages arrays of characters (strings). You can use any on-line reference to read more about string objects if needed. You will likely need the `length()` and `empty()` methods that give the length of a string and a boolean `true` value if the string is empty. Finally, you will create a `Print()` method in your `Complex` class to print the value of the complex number.

To parse the input strings for your complex calculator, a parser is provided. The parser takes an input line as input, and breaks it into substrings separated by numeric operators (+, -, * and /). Documentation of how to call the parser is included in the provided source code. If the first character of a substring is a left parenthesis, then all characters up to and including the subsequent right parenthesis is parsed as part of the substring. Thus the input:

```
(1,2) + 3
```

would be parsed as two substrings, the first is `(1,2)` with a "+" delimiter, and the second is the substring `3`. The provided skeleton program `calculator-skeleton.cc` demonstrates how to call the parser and prints out the parsed strings as debugging information. You should remove the debug code before finishing your program. Finally, the code given in `string-parse.cc` has two more useful functions. The first (`RemoveParens`) will remove leading and trailing parenthesis from a string. The second `ToDouble` will convert the string represenation of a floating point number to type `double`.

It is possible that a complex variable is *Not a Number* (*NaN*). This is the case when it is the result of a divide by zero, of when computing the angle of a zero magnitude complex value, or when the result of any operation where either of the operands are NaN. When printing a complex value that is *NaN*, the string "NaN" should be output.

A sample session is shown below:

```
(102,0)-(1,0)
 = 101
(1,2) + (3,6)
 = (4,8)
+(2,3)
 = (6,11)
+(0,  1)
 = (6,12)
/   (2,   0)
 = (3,6)
+5
 = (8,6)
(1>45)
 = (0.707107,0.707107)
/0
 = NaN
+10
 = NaN
1
 = 1
*(100,23)
 = (100,23)
```

There are a few more things you need to know.

1. The value of π is defined in "math.h" as the symbol `M_PI`. **DO NOT TYPE IN THE VALUE OF PI. Use teh `M_PI` defined constant instead.**
2. The trig functions you need are also defined in "math.h". In particular you might need `sin`, `cos` and `atan2`. All trig functions use *radians*, not degrees. The `atan2(y,x)` is *arc-tangent*. Since tangets can legally be infinity, `atan2` uses the form y/x and passes both y and x as arguments, getting around the possible problems with divide by zero.
3. The complex conjugate of a complex variable is another complex variable (r, -i). In other words, just negate the imaginary part.
4. The magnitude of a complex variable is just the square root of the real squared plus the imaginary squared.
5. To compute a/b where both a and b are complex values, do the following:
   (a) If the magnitude of b is zero, the result is *Not-a-Number*.
   (b) Compute a temporary variable c as the a times the complex conjugate of b.
   (c) Compute m as the magnitude of b squared.
   (d) The real part of the result of the division is c.real$/m$.real.
   (e) The imaginary part of the result is c.imag$/m$.real.

Copying the Project Skeletons

1. Log into `deepthought19.cc` using `ssh` and your prism log-in name.
2. Copy the files from the ECE2036 user account using the following command:

   ```
   /usr/bin/rsync -avu /nethome/ECE2036/ComplexCalculator .
   ```

   Be sure to notice the period at the end of the above command.
3. Change your working directory to `ComplexCalculator`

   ```
   cd ComplexCalculator
   ```

4. Copy the provided `calculator-skeleton.cc` to `calculator.cc` as follows:

   ```
   cp calculator-skeleton.cc calculator.cc
   ```

5. Then edit `calculator.cc` to implement the calculator.
6. Edit complex.h and complex.cc to implement your Complex number class.
7. Compile your code using `make` as follows:

   ```
   make
   ```

8. Once you have gotten the calculator program compiled and ready to test, you can just run it interactively:

   ```
   ./calculator
   ```

   and type in any calculator commands for debugging. Or you can use the "canned" inputs in input.txt as follows:

   ```
   ./calculator < input.txt
   ```

   The expected output matching "input.txt" is found in "output.txt".

Resources

1. `calculator-skeleton.cc` is a starting point for your program.
2. `complex.cc` and `complex.h` provide a skeleton C++ object for you to use as a starting point for your `Complex` class.
3. `Makefile` is a file used by the `make` command to build the calculator program.
4. `string-parse.h` and `string-parse.cc` are provided for you and are used to "parse" the calculator input into substrings. There are comments in both files describing their usage, plus we will discuss these in class.
5. `input.txt` is a set of inputs that the TA will use to "test" your program.
6. `output.txt` is the matching set of outputs for `input.txt`.

Turning in your Project. Use the turnin-ece2036a or turnin-ece2036b depending on which section you are in. Change your working directory to your "home" directory and use the turnin script as follows:

```
cd
turnin-ece2036a ComplexCalculator
```

Be sure to use turnin-ece2036b if in section B.