CSE 425 Studio: Semantic Analysis

These studio exercises are intended to give you familiarity with ideas and techniques for operational semantics, by implementing a simple reduction machine (for postfix complex number arithmetic expressions) in a multi-paradigm programming language (C++). In this studio you will again work in self-selected groups of ~3 people, and will report the results of your work on the studio exercises in an e-mail sent to the course account. Students who are more familiar with the material are encouraged to partner with and help those who are less familiar with it, and asking questions of your classmates and professor during the studio sessions is highly encouraged as well. There are also links on the main course web page to helpful web sites for these exercises, which you are also encouraged to use as resources. Please record your answers as you work through the following exercises. After you have finished please send e-mail your answers with "Semantic Analysis Studio" in the subject line, to the cse425@seas.wustl.edu course account. The enrichment exercise is a chance to dig deeper into the material, especially if you breeze through the required ones.

PART I: REQUIRED EXERCISES

1. Form a team of ~3 people and write down your names as the answer to this exercise.

2. On the Start menu in Windows, open up Visual Studio 2013 and create a new Win32 Console Application project for these studio exercises. Modify the main function signature so that it matches the standard portable (e.g., between Windows and Linux) main function signature (and variable names) for a C++ program: **`int main (int argc, char * argv[])`**.

In a separate header file, declare a class or struct (e.g., **`struct Complex`**) that implements a complex number. The internal representation is up to you (e.g., using Cartesian or polar coordinates), but if you used a class (or decided to represent the abstraction in terms of polar coordinates) you should provide public accessor methods for the real and imaginary parts of the complex number.

The public interface of the class or struct should include a constructor from double precision floating point numbers for the real and imaginary parts of the complex number, and a copy constructor. Also add a default constructor that initializes the object to be at the origin in the complex plane, e.g., (0,0). Outside of the class or struct declare an insertion operator that takes a reference to an ostream and a reference to a const complex number, and uses the ostream to output a representation of the real and imaginary parts of the complex number. In a separate source file, define all the methods and operators you declared.

In your main function construct, default construct, copy construct, and output different complex numbers. Build and run your program, and as the answer to this exercise show the code and the output the program produced.

3. Declare and define binary addition and multiplication operators (**`operator+`** and **`operator*`**) for your complex number class or struct. These can be either members of the class or struct, or external ones declared and defined outside the class or struct.

In your main function, print out the results of expressions that add and multiply different combinations of the objects from the previous exercise. Build and run your program and as the result of this exercise show the definitions of the addition and multiplication operators, your main function with the expressions you tested, and the output your program produced.

4. Outside your main function, write a simple classification function (feel free to use or adapt code from previous studios for this) that can identify each of the command line arguments that were passed to the program (starting with **`argv[1]`**) as either a number (all numeric digits), an addition operator (**`"+"`**), a multiplication operator (**`"*"`**), or something else.

Your program should go through the arguments in **`argv`** (starting with **`argv[1]`**), repeatedly looking for two numbers in a row, treating the first one as the real part and the second one as the imaginary part, and using them to construct a complex number object and print it out, and then (starting at the next position after them) looking for another adjacent pair of numbers. For example, if run as

```
SemanticAnalysisStudio.exe hello 7 9 8 world 10 14
```

the program should print out the complex numbers **`(7, 9)`** and **`(10, 14)`**. Build your program and run it with different command line arguments to confirm it picks out adjacent pairs of numbers and constructs valid complex number objects from them. As the answer to this exercise show a few representative command lines you tried, and the output that the program produced for each of them.

5. Modify your code from the previous exercise so that whenever it detects a complex number it pushes it onto a vector (or other STL sequence container) that will act as a reduction stack for these exercises, and prints out a message saying that it has pushed that complex number onto the reduction stack.

Whenever the program sees an addition operator (**`"+"`**) it should try to pop two complex number objects off the reduction stack (and should ignore all other kinds of strings in **`argv`** besides numbers and addition operators). If it fails to pop two numbers off the reduction stack the program should print out an error message saying that it was unable to continue processing, and should end. Otherwise, the program should use the complex number class addition operator to add the two complex numbers it just popped, and then push the resulting complex number back onto the reduction stack. Whenever it does that it should print out a message with the two complex numbers it popped off the stack, the addition symbol, and the resulting complex number it pushed back onto the stack.

Build your program and run it with well formed postfix addition expressions like **`8 7 6 5 + 9 0 +`** as well as badly formed ones like **`8 7 6 5 + +`**, and as the answer to this exercise please show a few representative command lines you tried, and the output that the program produced for each of them.

6. Modify your code from the previous exercise so that whenever the program sees a multiplication operator (**`"*"`**) it should try to pop two complex number objects off the reduction stack, and if it fails to do so the program should print out an error message saying that it was unable to continue processing, and

should end. Otherwise, the program should use the complex number class multiplication operator to push the product of the two complex numbers it just popped, back onto the reduction stack. Whenever it does that it should print out a message with the two complex numbers it popped off the stack, the multiplication symbol, and the resulting complex number it pushed back onto the stack.

Build your program and run it with different well formed postfix expressions involving both addition and multiplication like **`8 7 6 5 + 9 0 *`** as well as badly formed ones like **`8 7 6 5 + *`**, and as the answer to this exercise please show a few representative command lines you tried, and the output that the program produced for each of them.

PART II: ENRICHMENT EXERCISES (Optional, feel free to skip or try variations that interest you)

7. Extend your code from the previous exercises to support a binary subtraction operator (**`operator-`**) in your class or struct. Also modify your code so that whenever the program sees a subtraction operator (**`"-"`**) it should try to pop two complex number objects off the reduction stack, and if it fails to do so the program should print out an error message saying that it was unable to continue processing, and should end. Otherwise, the program should use the complex number class subtraction operator to subtract the ***first*** complex number it popped off the stack from the ***second*** one, and push the result back onto the reduction stack. Whenever it does that it should print out a message with the two complex numbers it popped off the stack, the subtraction symbol, and the resulting complex number it pushed back onto the stack.

Build your program and run it with different well formed postfix expressions involving addition, multiplication, and subtraction, like **`8 7 6 5 * 9 0 + 4 1 -`** as well as running it with badly formed ones like **`8 7 6 5 * 9 0 + -`**, and as the answer to this exercise please show a few representative command lines you tried, and the output that the program produced for each of them.

8. Extend your class with a division operator (**`operator/`**). Modify your code from the previous exercise so that whenever the program sees a division operator (**`"/"`**) it should try to pop two complex number objects off the reduction stack, and if it fails to do so the program should print out an error message saying that it was unable to continue processing, and should end. Otherwise, if the first complex number it popped off the reduction stack is (0, 0) the program should print out a different error message saying that it was unable to continue processing (due to division by zero), and should end. Otherwise, the program should use the complex number class division operator to divide the ***second*** complex number it popped off the stack by the ***first*** one, and push the result back onto the reduction stack. Whenever it does that it should print out a message with the two complex numbers it popped off the stack, the division symbol, and the resulting complex number it pushed back onto the stack.

Build your program and run it with different well formed postfix expressions involving addition, multiplication, subtraction, and division like **`8 7 6 5 * 9 0 + 3 1 - 2 4 /`** as well as running it with badly formed ones like **`8 7 6 5 * 9 0 + 3 1 - /`**, or ones that would divide by zero like **`8 7 6 5 * 9 0 + 3 1 - 0 0 /`**, and as the answer to this exercise please show a few representative command lines you tried, and the output that the program produced for each of them.