Assignment 6

Due May 28, 2015 by 11:59pm **Points** 100 **Submitting** a text entry box or a file upload
Available after May 19, 2015 at 3:30pm

Stacks and Compilers

Purpose

This programming assignment continues to extend the feature of the textbook's calculator example. You will implement unary operators and the operator= in your calculator. Furthermore, your calculator must be able to handle three variables such as a, b, and c.

Unary Operators

Those includes '-', '+', '!', and '~'. For those operators' precedence, refer to http://www.cppreference.com/operator_precedence.html (http://www.cppreference.com/operator_precedence.html) . Among them, a special attention must be paid to '-' and '+', because you have to decide if they are binary or unary operators. You can make a decision in tokenizer.cpp.h's getToken( ) as follows:

1. Prepare **TokenType prevToken** as a private data member of **tokenizer.h**.
2. Whenever **tokenizer.cpp.h** returns a toke type, memorize it in **prevToken**.
3. When encountering '+' or '-', your **tokenizer.cpp.h** must check **prevToken** if the previous token was '('. If so, you can return the unary '+' or unary '-'. Otherwise, you should return the binary '+' or binary '-'.
4. Note that **prevToken** must be initialized with OPAREN (i.e, '(') so that your **tokenizer.cpp.h** can accept the very first '+' or '-' as a unary operator even in case if they do not follow an open parenthesis.

Variables

Your calculator is supposed to handle only three variables such as a, b, and c. While the actual implementation is up to you, the easiest (but an unextended) implementation is as follows:

1. **evaluator.h**: add the following private data members.

```
private:
 // new members
 vector postFixVarStack;            // Postfix machine stack for var
 NumericType var_a;                       // variable a
 NumericType var_b;                       // variable b
 NumericType var_c;                       // variable c
```

2. **token.h**: add the following special tokens, each corresponding to variables **a**, **b**, and **c.**

```
enum TokenType {
  ...
  VAR_A,       // variable a
  VAR_B,       // variable b
  VAR_C        // variable c
};
```

3. **tokenizer.cpp.h**: return the corresponding token when getToken( ) encounters 'a', 'b', and 'c.

```
  if ( getChar( ch ) == true ) {
    switch( ch ) {
    ...
    case 'a':
      prevToken = VAR_A;
      return Token<NumericType>( VAR_A, 0 ); //updated 5/26/2015
    case 'b':
      ... // the same as 'a'
    case 'c':
      ... // the same as 'b'
    ...
    default:
      ...
    }
  }
```

4. **evaluator.cpp.h**: add three **case** statements, each corresponding to **VAR_A**, **VAR_B**, and **VAR_C** and pushing the variable content in **postFixStack** and the variable name in **postFixVarStack.** For a constant value, push the constant value to **postFixStack** as you did previously and push a space ' ' to **postFixVarStack.**

```
  switch( lastType ) {
  case VALUE:
    postFixStack.push_back( lastToken.getValue( ) );
    postFixVarStack.push_back( ' ' );
    return;

  case VAR_A:
    postFixStack.push_back( var_a );
    postFixVarStack.push_back( 'a' );
    return;

  case VAR_B:
    ... // the same as VAR_B
  case VAR_C:
    ... // the same as VAR_C
```

Whenever **evaluator.cpp.h** needs to pop out a value from **postFixStack**, you must also pop out a character from **postFixVarStack.** Similarly, whenever **evaluator.cpp.h** needs to push a new value to **postFixStack**, you must also push the corresponding character ('a', 'b', 'c', or ' ') to **postFixVarStack.** This way allows both **postFixStack** and **postFixVarStack** synchronously grow and shrink.

Assignment Operators

Although C++ assignment operators include =, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, and >>=, we focus on only '='.

Statement of Work

1. Download cpp_evaluator1.zip to your project. Replace the evaluator.cpp.h and tokenizer.cpp.h with the ones from Lab 6.
2. Implement unary operators '+', '-', '!', '~', an assignment operator '=', and variables 'a', 'b', and 'c'. You may follow the above instructions or develop your own algorithm.
3. Compile all your code.
4. Execute this **evaluator** program to verify your modification.

```
$ ./a.out
8 + (-5 + 2)
5
8 + (+5 + 2)
15
-5 + 8 + 2
5
+5 + 8 + 2
15
!(3 == 3)
0
!3 == 0
1
~2
-3
a = 10 * 3
30
b = 5 * 2
10
c = 8 % 3
2
(a = 10 * 3) + (b = 30 / 3) + (c = 50 - 25) + a + (-b) + ~c * 3
7
^c (to exit the  program)
```

What to Turn in

Clearly state in your code comments any other assumptions you have made.

Your soft copy must include:

(1) all .h and .cpp files

(2) execution outputs in a .doc, .docx, or text file.

Grading Guide and Answers

Click the following grading guide to see how your homework will be graded. Key answer will be made available after the due date through Solution page.

```
Program 6 Grade Guideline

1. Documentation (10pts)
   Execution outputs
   Satisfactory outputs (10pts)            Unsatisfactory outputs  (5pt)
   No outputs (0pt)

2. Correctness (70pts)
   Successful compilation
   Compiled(20pts)    Compilation errors(0pt): If not compiled, 0 for this correctness category

   Unary operators + - ! and ~
   Correct(25pts)  1 bug(20pts)    2 bugs(15pts)    3 bugs(10pts)    4+ bugs(0pt)

   Assignment operator= and variables a b and c

   Correct(25pts)  1 bug(20pts)    2 bugs(15pts)    3 bugs(10pts)    4+ bugs(0pt)
```

```
3. Program Organization (20pts)
   Write comments to help the professor or the grader understand your operations.

   Proper comments
   Good (10pts)       Fair (7pts)      Poor(3pts)        No explanations(0pt)
```

```
Coding style (proper identations, blank lines, variable names, and non- redundant code)
   Good (10pts)       Fair (7pts)      Poor(3pts)        No explanations(0pt)
```