ENSC 251 – Summer 2017 – Course Project Part 2

In the first part of the course project, you have implemented the first step of a parser for a subset of the C++ language, i.e., a basic tokenizer that can split a stream into a sequence of strings representing tokens for the C++ language subset. In this part, you will be implementing a set of classes into which the tokens of the C++ language subset can be categorized according to different token types. Unlike the first part of the project in which all the tokens were stored as strings, in this part the tokens are constructed as objects of corresponding classes. This may seem like a lot of work now, but the classes will be useful when we get to the next parts of the project where we construct an abstract syntax tree and then go on to interpret the parsed C++ language subset program.

For example, the below classification would be generated for the following two lines of code:

int sum;

Token:	int	Sum	;
Class:	type_spec	int_id	punctuator

sum = 10+2.5;

Token:	sum	=	10	+	2.5	;
Class:	int_id	gen_assignment_operator	numeric_int_const<int>	additive_operator	numeric_const<float>	punctuator

NOTE 1: the identifiers of type 'char' and 'int' should be classified into the 'int_id' class, the identifiers of type 'float' should be classified into 'numeric_id', and the pointers such as 'char*' should be classified into the 'pointer_id' class.

An eclipse project template including the required classes (and their corresponding members for some classes) and the signature for the token classifier function has been provided. You may implement any desired helper functions for the classifier, but you are not allowed to remove the classifier function or any classes from the provided program structure or change its overall behavior. Please do not modify the names of any of the provided files.

Similar to what we have done in Part 1, the input to the "tokenClassifier" function in tokenClassifier.cpp is an istream object. Below is the class hierarchy for classifying tokens into specific objects.

```
Token
  |_ StringBasedToken
  |        |_ punctuator
  |        |_ type_spec
  |        |_ gen_assignment_operator
  |        |_ int_assignment_operator
  |        |_ conditional_operator
  |        |_ shift_operator
  |        |_ additive_operator
  |        |_ div_operator
  |        |_ mod_operator
  |        |_ comp_operator
  |        |_ postfix_operator
  |        |_ string
  |        |_ incorrect
  |        |_ id
  |            |_ pointer_id
  |            |_ numeric_id
  |                       |_ int_id
  |_ constant
           |_ numeric_const <T>
                     |_ numeric_int_const <T>
```

Note that numeric_const <T> class (which is inherited from class *constant*) is a template class with typename T. Depending on the type of the constant token, one can instantiate an object for **float** using the **numeric_const**, and **int** or **char** using the **numeric_int_const** class templates. To get a preview on Templates, please refer to Chapter 17 of our Textbook.

Take care with floating point numbers. For example, 1.575E1 is equivalent to 15.75. So, "1.575E1" is one token. This token during classification would be stored as an object of class **numeric_const <float>**.

NOTE 2: When testing your code we may use some tests that are correct according to our C++ subset grammar, some tests that include things not present in the grammar (like operators or punctuators not present in the grammar), and some tests that have other sorts of errors (like unterminated strings). During Part1, we asked you to push an empty string onto the back of the vector of strings when an invalid token was encountered, and then continue on tokenizing the remaining characters. Now in this part of the project, output to the console a warning message and push a pointer to an "incorrect" object (which stores the unclassifiable characters) onto the vector to be returned when such unclassifiable characters are encountered.

NOTE 3: Each token object is also required to store the line number and the starting index number in which the token is located in the input.

For example, for the input

"errno=1;\nerrno++;"

the following line and index numbers should also be stored (errno is a global int declared in the header file <errno.h> and should be classified as an int_id):

Token	**Line number**	**Index number**
errno	0	0
=	0	5
1	0	6

;	0	7
errno	1	0
++	1	5
;	1	7

Here is a longer example that we will be considering:

```
int a;
int b;
int c;
char *e;
float g;
a = 2;
b = a ? a : 4;
c = 4 + b / (1 + a);
e = "Hello World, Goodbye World";
a <<= 2;
g = 1.2 + 4;
```

INSTRUCTIONS FOR SUBMITTING YOUR CODE:

- Remember, do not modify the names of files given to you.