CSCE2014 Programming Foundations II
Homework Three
By Wing Ning Li

1 Problem Description

In our lab 4, we worked on classes as a review. In particular, we designed two classes Song and MusicLibrary and implemented them partially. This assignment will develop two more classes: *Token* and *Expression*. We wish to use them in our last homework assignment. In computer science, tokens represent basic elements or building blocks in a complex statement or expression. The actual token definition varies from applications to applications. C++ string has member functions that support tokenization and C string has library functions such as strtok (which we used in lab 3 and implemented in homework Two). Sometimes due to no delimiter between tokens, we have to write code to tokenize as in this homework.

Let us look at a few examples of expressions and identify the tokens involved. Please pay close attention to the spaces and the lack of spaces in the expressions. In `1+2`, we have three tokens. They are `1`,`+`,`2`. In `a=5`, we have again three tokens. They are `a`, `=`, `5`. In `(a + 123 )*(ab - (3 + 4 ))`, we have fifteen tokens. They are `(`, `a`, `+`, `123`, `)`, `*`, `(`, `ab`, `-`, `(`, `3`, `+`, `4`, `)`, `)`. Here is another example of an expression, which is invalid in several ways, and its tokens. In `a12 = 1?ab + - a 0123 c (a + 12 3 )*(ab - (3 + 4 ))`, we have 24 tokens. They `a12`, `=`, `1?ab`, `+`, `-`, `a`, `0123`, `c`, `(`, `a`, `+`, `12`, `3`, `)`, `*`, `(`, `ab`, `-`, `(`, `3`, `+`, `4`, `)`, `)`.

Based on the examples above, we have two categories of tokens: *special tokens* and *ordinary tokens*. **Special tokens are the following single character tokens:** `(`, `)`, `+`, `-`, `*`, `/`, and `=` (open and close parentheses, arithmetic operators, and the assignment operator). **Ordinary tokens are any contiguous sequence of printable characters separated by space (one or more blank characters)or special tokens.** Furthermore, tokens may be *valid* or *invalid*. All special tokens are valid. Ordinary tokens are valid if the tokens are nonnegative integers (e.g. 0, 7, 16, 777) or begin with a letter followed by any number of letters or digits (identifiers). Ordinary tokens that do not satisfy the above are invalid. In the last example above, ordinary tokens `a12`, `a`, `c`, `12`, `ab`, `3`, and `4` are valid tokens. And ordinary tokens `1?ab` and `0123` are *invalid* according to the definition above.

Two types of expressions are illustrated in the examples above: *Assignment Expression* and *Arithmetic Expression*. An assignment expression has three parts: an identifier (an alphanumerical string that begins with a letter), an equal sign, and a nonnegative integer. An arithmetic expression involves binary operators for addition, subtraction, multiplication, and division; operands (nonnegative integers and identifiers); and parentheses for grouping. Arithmetic expressions are evaluated and constructed based on the rules we learned from grade school to high school.

2 Purpose

Understand class, member and member function, and constructor. Learn and know how to use **string** (C++ string) and **vector**, which may be viewed as our first example of a template class and a container class. Understand the concepts of token and expression. Learn how to tokenize an expression.

3 Design

3.1 Token Class

The Token_type, which is an enum type and used in the *Token* class, is defined as follows:

```
enum Token_type {ID, INT, OP, EQ, OpenBrace, CloseBrace, INVALID};
```

If string s, passed in to the constructor or set method, is an identifier then the token type is ID; is a nonnegative integer then the token type is INT; is one of `+`, `-`, `*`, `/` then the token type is OP; is `=` then the token type is EQ; is `(`, open parenthesis, then the token type is OpenBrace; is `)`, close parenthesis, then the token type is CloseBrace; is none of the above then the token type is INVALID. The constructor and set method are required to figure out the type of a token string.

Token class is used to store a "token", which has three members and a few member functions. The members are:

type This field stores the type of the token. Its type is Token_type whose definition is given earlier.

token This field stores the token. Its type is string.

priority This field stores the priority of the token when token is an operator or a parenthesis. Its type is int.

The member functions are:

Token() This default constructor initializes *type* to INVALID, *token* to empty string, and *priority* to -1.

Token(string s) This constructor takes in a string and treats it as a token. It sets the type and token member accordingly. For the current project, it sets the priority member to -1.

void set(string s) This method takes in a string and treats it as a token. It sets the members based on the new token s. Again, for the current project, it sets the priority member to -1.

int value() const This method returns the value associated with the token when token type is integer or identifier. In this assignment, if the type is INT, the integer represented as a string stored in token member should be converted to int and returned; if the type is ID, it returns -1; for all other types, it returns -2.

In the future assignment, we will be able to process assignment statement such as x=5 and a symbol table is used to associate x with 5. In this case, for token x, the function returns 5.

void display() const This method outputs the values of the token members in three lines. Each line begins with the member name, followed by "=", followed by the value. This method is mainly for debugging use. For example, if token string is `a12`, we have:

```
type = ID
token = a12 (value is -1)
priority = -1
```

Token_type get_type() const Getter for type member.

string get_token() const Getter for token member.

int get_priority() const Getter for priority member.

Typically during the tokenizing process of the input string, we get a sequence of characters stored in a string, which is a token. We then use *Token* class via its constructor or set method to figure out if the token is valid or not and what type of token it is and so on. **That is perhaps the most challenging programming (algorithmic) task for the implementation of the *Token* class.** Since an expression is really a sequence of tokens in an abstract sense, the *Token* class is used by the *Expression* class to be designed next.

Since none of the member of the *Token* class involves pointer type and memory allocation, the default *copy constructor* and *assignment operator* provided by the compiler should work just fine. We do not need to implement them.

3.2 Expression Class

The Exp_type type, which is an enum type and used in the *Expression* class, is defined as follows:

```
enum Exp_type {ASSIGNMENT, ARITHMETIC, ILLEGAL};
```

For the current assignment, we are not required to figure out the type of an expression. We will do that in a later assignment. For the current project, we treat an expression in two ways: 1) as a string and 2) as a sequence of tokens (Token objects stored in a `vector<Token>`) obtained from the string.

Expression class is used to store an "expression", which has five members and a few member functions. The members are:

original This field stores the original or not yet processed "expression". Its type is string.

tokenized This field stores the expression as a sequence of tokens. Its type is `vector<Token>`.

postfix This field stores the expression as a sequence of tokens in postfix notation provided the expression has arithmetic type. Its type is `vector<Token>`. **This member is not used for the current homework assignment but will be used for the last assignment. The set method or constructors of this assignment should leave it as an empty vector, that is to do nothing**.

valid This field indicates whether the expression is a valid expression (no syntax error). Its type is bool. **It is not used for the current homework assignment but will be used for the last assignment. The set method or constructors of this assignment should simply set it to false.**

type This field indicates whether the type of the expression is *ASSIGNMENT* or *ARITHMETIC* expression or *ILLEGAL*. Its type is Exp_type. **It is not used for the current homework assignment but will be used for the last assignment. The set method or constructors of this assignment should simply set it to ILLEGAL.**

The member functions are:

Expression() This default constructor initializes all fields to empty or false or ILLEGAL.

Expression(const string& s) This constructor takes in a string and treats it as an expression. It tokenizes the input string and sets the `original` and `tokenized` members accordingly. And it sets the other members based on the description given earlier (refer to each member description).

void set(const string& s) This method takes in a string and treats it as an expression. It tokenizes the input string and sets the original and tokenized members accordingly. And it sets the other members based on the description given earlier.

void display() const This method output the values of the expression fields, one field per line. This method is mainly for debugging use.

```
original = a12 = 1?ab  +  - a 0123 c (a + 12 3 )*(ab - (3 + 4   ))
tokenized = a12;=;1?ab;+;-;a;0123;c;(;a;+;12;3;);*;(;ab;-;(;3;+;4;););
number of tokens = 24
postfix =
valid = false
type = ILLEGAL
```

string get_original() const Getter for original member.

vector<Token> get_tokenized() const Getter for tokenized member.

other methods There are other methods such as to evaluate the expression, to transform the expression, and getters for other fields such as postfix, valid, or type. These methods will be needed for the last assignment. Since we have not studied the related concepts and algorithms, these member functions are omitted.

When we receive a string (maybe from a line of input or a hard coded value within the program), which would be an expression, we use *Expression* class to figure out if the expression is legal or not and what its type is and so on (use either the non default constructor or set method, and pass the string as its argument).

In order to do the above, the string (perhaps a line of input from the user) gets tokenized first by the *Expression* class via its constructor or set method. **That is perhaps the most challenging programming (algorithmic) task for the current implementation of the *Expression* class.**

Since none of the member of the *Expression* class involves pointer type and memory allocation, the default *copy constructor* and *assignment operator* provided by the compiler should work just fine. We do not need to implement them.

4 Implementation

Please note the similarity between the constructor that takes a string parameter and the set method in *Token* class. We could implement the set method first. Then the constructor calls the set method to accomplish its tasks. The same can be said for the similarity between the non default constructor and set method in *Expression* class.

You should implement and test the *Token* class first. Then use it in the development of the *Expresion* class. The most challenging method to implement (develop the code for) is the set method of the *Expression* class. We need to figure out the right algorithm to tokenize. One of the reasons is that we cannot simply use space (a blank character) as a delimiter to tokenize. For instance, in `(2+3)*4`, we have 7 valid tokens. They are `(`, `2`, `+`, `3`, `)`, `*`, `4` and no space between them. But if it is given as `( 2 + 3 ) * 4`, we could use space. Our method should work for any possible combination of spaces and tokens. You might first assume space is used to separate each token including special tokens and develop a solution. Then consider if you might add space to the given string to use your with space as delimiter solution earlier. This approach is perhaps less efficient than those algorithms that process the string once because it processes a string twice and might double its length. However, it is acceptable for this assignment.

For those students who would like a challenge, you might develop an efficient algorithm that tokenizes the string with one pass through the given string.

Token class implementation has .h and .cpp files and the same is true for Expression class. The homework3.cpp file contains the code for testing of both classes.

5 Test and evaluation

Have a main program to test the *Token* class by sending the constructor and set method many different "tokens" and display the results to see if they are correct or not. You may want to be creative and cover a wide range of possible "tokens" in your testing. Similarly, test the *Expression* class by sending the constructor and set method many different "expressions" and display the results to see if they are correct or not. You may want to be creative and cover a wide range of possible "expressions" in your testing.

6 Report and documentation

A short report about things observed and things learned and understood. The report should also describe the test cases used in the main program and the reasons for each test case selected. Properly document and indent the source code. The source code must include the author name and as well as a synopsis of the file.

7 Homework submission

Use Blackboard to submit all your source code files (.h and .cpp files for Token class, for Expression class, and homework3.cpp) and the report in either .pdf or .doc or .docx format.