CSE 465/565

Homework #6: (80) points

Outcomes:

- Working with C++ as one of Imperative Programming Languages
- Understanding how C++ works and differences with other imperative languages
- Working with some of C++ features
- Working with files in C++
- Solving problems using C++
- Running and testing C++ codes

Scoring (465/565):

- (5 points) submission on Git with **10 or more commits**. Your Git history should represent the progress of your work.
- (60 points) Write an interpreter using C++ language to execute Z+- (.zpm) programs
    - (5/5 pts) Variables and integer values (test1.zpm)
    - (5/5 pts) Variables and String values (test2.zpm)
    - (5/5 pts) Print (test1.zpm & test2.zpm)
    - (5/5 pts) Variables assignments (test3.zpm)
    - (5/5 pts) Operators (test4.zpm)
    - (10/5 pts) For loops (test5.zpm)
    - (-/5 pts) Nested for loops (grad.zpm)
    - (5/5 pts) Runtime error (test6.zpm)
    - (20/20 pts) An actual Z+- program (actual_program.zpm)
- (5 points) Using pointers as part of your solution
- (5 points) Your interpreter should have no memory leak
- (5 points) Quality and Style of your code including naming, comments, length of methods, etc.
- (5 bonus points) Achieve a very good elapsed time on 'sample_bonus.zpm'.

Requirements:

- On your laptop, add a new folder inside your **CSE465_565** folder, and call it **Homework6**.
- Your main file **Zpm.cpp** should be saved inside the **Homework#6** folder.
- **There will be no partial credit for this assignment. Your code must either generate correct results for each .zpm file, in which case you will receive full points for that file, or you will receive zero point for that file if there is even one wrong value in the results.**

Instructions:

1. In your project, you can have as many classes as you like either in one file or more than one file. However, you must have at least one class called **Zpm.cpp**, which will be run by your instructor.
2. Your program must take a .zpm file as a command line argument. Use the following command to test your code:

   ```
   g++ -std=c++11 -o Zpm Zpm.cpp
   ./Zpm test1.zpm
   ```

   The above example is for the case where you only have one file called Zpm.cpp. If you have more than one file, it's necessary to mention those files during compilation. Here is an example for compiling and running where there are two files:

   ```
   g++ -std=c++11 -o Zpm Zpm.cpp Helper.cpp <=OR=> g++ -std=c++11 -o Zpm *.cpp
   ./Zpm test1.zpm
   ```

3. Every statement is terminated by a semi-colon. Except lines that end with the ENDFOR.

4. You may assume that the programs are syntactically correct.

5. There might be empty lines in the .zpm files, which should be ignored.

6. Z+ variables are case-sensitive and consist of one or more letters and digits. Variables can include both lowercase and uppercase letters. Variable names do not start with digits.

7. Z+- variables can store a string or integer value. Assigning a value to a variable creates that variable for future use.

8. In Z+- language we can print the value of the variables, and it prints them on the display with the following format:

   <variable's name>=<variable's value>

   There is no space before or after = sign. For example:

   ```
   a = 10 ;
   s = "Hello World!" ;
   PRINT a ;
   PRINT s ;
   ```

   The result:

   ```
   a=10
   ```

s=”Hello World!”

9. The right-hand side of a simple assignment statement, using = operator, is either an integer or String variable (which must have a value), or an integer or String value. A single variable can switch between integer and string values during program execution.

   <variable name> = <variable which has a value/value>

10. There are three compound assignment statements: +=, *=, and -=. The meaning of these operators depends on the data type of the left and right hand side of the operator.

    <string var> += <string var/val> concat right string variable/value onto end of left string
    <integer var> += <integer var/val> increment left integer with value/value of a variable on right
    <integer var> *= <integer var/val> multiply left integer by value/value of a variable on right
    <integer var> -= <integer var/val> subtract right integer from value/value of a variable on left

11. There is a loop statement whose body contains at least one simple statement (i.e., no nested loops), which are presented on one line.

    FOR <integer var/val> <statement/s separated by semicolon> ENDFOR

    The keyword FOR is followed by an integer value or variable, which indicates the number of times to execute the loop. Following this number is a sequence of statements (at least one statement) separated by semicolon, defining the loop's body, followed by the word ENDFOR. The following are acceptable FOR loops statements:

12. Commit and push your work right after any small progress. Your Git should demonstrate the progression of your work.
    - Adding or removing empty lines or spaces in one commit, does not add authenticity to your work.
    - Committing and pushing large volumes of code within a very short time frame, such as minutes or seconds apart, can raise doubts about the work's originality. Producing significant amounts of code this quickly is unrealistic and may indicate that the code is copied from another source. Genuine coding usually requires more development time and deliberate, thoughtful submission of changes.
13. Your program should be able to catch only one runtime error, which happens when different types of values or variables are used in calculation (using operators). Your program should print an error with the following format and stops:

    RUNTIME ERROR: line <the number of the line with the error>

    Example1 of a Z+- code with an error:

    ```
    ----------
    A = 0 ;
    B = “hello” ;
    A += B ;
    PRINT A ;
    ----------
    ```

The result:
RUNTIME ERROR: line 3

Example2 of a Z+- code with an error:

B = “hello” ;
B += 12 ;
PRINT B ;

The result:
RUNTIME ERROR: line 2

Example3 of a Z+- code with an error:

A = -10 ;

A -= “ ” ;
PRINT A ;

The result:
RUNTIME ERROR: line 4

**** Empty lines must be counted ****

Bonus Points:

To earn 5 bonus points on your assignment, ensure your code achieves a competitive runtime. Your runtime will be compared against others, and if it ranks among the top 15 (with some margin of error), you will receive the full bonus points. At the beginning of your **Zpm.cpp** file, add a special comment indicating whether you want your runtime to be compared for bonus points. Use the following format:

// => I’m competing for BONUS Points <=

- Failing to highlight your participation to achieve bonus points by not adding the required comments at the top of your **Zpm.cpp** file will result in 0 bonus points.

By default, these 5 bonus points will be added to your overall Homework grade. However, you have the option to request that these points be applied to either your Quizzes or Exams instead. Please note, these points cannot be applied to Attendance. To make this request, simply email your instructor specifying where you would like your bonus points to be added.

Test your program:

Come up with more tests and test your program comprehensively. Your code will be tested separately with different .zpm files.

Proper testing requires time. Begin by considering various scenarios and delving into the specifics. Eventually, you'll need to come up with a way to validate your solutions using the data at hand. Remember, thorough testing demands quality time devoted at the end to ensure your code functions as intended. So, as you manage your schedule for this assignment, make sure to allocate sufficient time for testing. The consequence of lack of proper testing will result in losing points.

Submission:

Submit the GitHub/GitLab URL of the project **Homework6**.
Inside this folder there should be the following files:

1. **Zpm.cpp** and other complimentary classes.
2. All the zpm test files.