School of Computer Science, McGill University

COMP-206 Introduction to Software Systems, Winter 2020

Mini Assignment 6: C Programming - Dynamic Memory

Due Date April 14, 23:55 EST

This is an individual assignment. You need to solve these questions on your own. Use the discussion forum on Piazza if you have any questions. Late penalty is -5% per day. Even if you are late only by a few minutes it will be rounded up to a day. Maximum of 2 late days are allowed.

This assignment will be worth 8% of your overall grade, unlike the other assignments that were worth only 4%.

You MUST use `mimi.cs.mcgill.ca` to create the solution to this assignment. You must not use your Mac command-line, Windows command-line, nor a Linux distro installed locally on your laptop. You can ssh or putty from your laptop to `mimi.cs.mcgill.ca`. All of your solutions should be composed of commands that are compilable/executable in `mimi.cs.mcgill.ca`.

Questions in this exercise requires you to turn in a number of files (will be listed in the description below). Instructors/TAs upon their discretion may ask you to demonstrate/explain your solution. No points are awarded for commands that do not execute at all or programs that do not compile in `mimi`. (Commands/programs that execute/compile, but provide incorrect behavior/output will be given partial marks.). All questions are graded proportionally. This means that if 40% of the question is correct, you will receive 40% of the grade.

Please read through the entire assignment before you start working on it. You can loose up to 3 points for not following the instructions.

Unless otherwise stated, all the names of the scripts and programs that you write, commands, options, input arguments, etc. are implied to be case-sensitive.

Total Points: 20

Ex. 1 — A Polynomail Evaluation App

In this exercise, you will create a C modules based program, `polyapp` to implement a simple application that constructs a polynomial expression and evaluates it for certain predetermined values.
The `polyapp` application will receive a text file as input, that contains multiple lines of data to construct a polynomial. Each line will contain the *coefficient* and *exponent* of a term in the polynomial. For example, below is an example file that represents the polynomial $12x^3 - 4x^5$.

```
$ cat data.ssv
12 3
-4 5
```

Here 12 and 3 are the coefficient and exponent of the term $12x^3$, respectively, and so forth.
You can use vi to create a data files of your own for testing purposes.

1. Create a directory for this assignment and initialize with git (just a local repository, do not make it public). As you progress through this assignment, you will keep commiting code to git (I leave the actual points in development where you feel like you should commit the code to git up to you, but **there should be at least 4 commits separated out by 15 minutes or more**).
2. All your program files should include a comment section at the beginning that describes its purpose (couple of lines), the author (your name), your department, a small "history" section indicating what changes you did on what date. The code should be properly indented for readability as well as contain any additional comments required to understand the program logic.

3. Your main will be in a C file `polyapp.c`. Your program will be invoked as follows by passing a data file as its argument. (Therefore, do not assume the name of the data file, but read it from the argument).

```
$ ./polyapp polydata.csv
```

   Although mini tester will not do any negative test cases by invoking the program with no file names as argument etc., I highly encourage you to include such logic in your program so that it is easy for you to develop and debug your program.

4. In your main, use `fgets` to read one line at a time from the data file.

5. Use vi to create a C file, `utils.c`.
   Create a function `parse` that accepts a string, and two integer pointers as its argument. Your main will use this `parse` function to parse a line that was retrieved using `fgets` and store the coefficient and exponent to the address provided in the integer pointers.
   Create a function `powi` that accepts two integers as argument (x and exponent) it returns an integer which is x raised to the power of exponent. For example, if it is passed 2 and 3, it will return 8, which is 2^3. You **must not use** `math.h` to perform this. Write your own simple loop to do the computation.

6. Create a C file `poly.c`. Here you will make a linked list that can be used to store a polynomial of arbitrary length. Your linked list must use the following structure for its nodes.

```
struct PolyTerm
{
  int coeff;
  int expo;
  struct PolyTerm *next;
};
```

   You will need a global variable, `head`, in this C file to keep track of the polynomial linked list. This C file will need a minimum of the following three functions (you are recommended to have more, as needed, to make your program modular and easy to develop and debug).
   `addPolyTerm` is a function that accepts two integer arguments (a coefficient and exponent) and assimilates that term into the existing Polynomial. It returns an integer as its return value. A 0 return indicates success and a -1 for failure (as in for example, ran out of memory, so could not add a new node to the polynomial linked list).
   `displayPolynomial` is a function that accepts no arguments and has no return. It will display the polynomial expression (will see the format later).
   `evaluatePolynomial` is a function that accepts an integer value for `x` and returns an integer that is obtained by evaluating the polynomial. This function will have to traverse the linked list, getting the values for coefficient and exponent from each node and using the `powi` function to perform computations.

7. Have your main parse each line of the data file to extract the coefficient and exponent and add it to the polynomial using the `addPolyTerm` function. If you run out of memory (the tester has no test cases for this), print an appropriate message and terminate your program from the main.

8. When all the polynomial terms are read and the polynomial is fully constructed, the main should call `displayPolynomial` to print the polynomial expression to the screen (format will be specified below).

9. Next, the main should call `evaluatePolynomial` for the values (-2, -1, 0, 1, -2) in that order and print the results (format will be specified below).

10. You may need additional header files (`.h`) to make sure your various C files can interact with each other. Create them as necessary and include them in appropriate files as needed. The objective is that `make` / `gcc` should not give any compilation warnings and errors.

11. Create a make file, name it `Makefile` . This make file should contain necessary compilation steps to build your final executable, which should be named `polyapp`. Your program should build an executable if TAs download your source code into an empty directory and just typed `make` . Therefore, make sure whatever commands you include in your make file is not your custom scripts, aliases, etc., but regular commands available to everyone in `mimi`.

Point distribution

1. **(1 Point)** For writing comments on all of your files.

2. **(2 Points)** Turn in your git log, by redirecting its output and store it to a file `gitlog.txt`. It should show **at least 4 commits which are separated by at least 15 minutes or more**.
3. **(2 Points)** For compiling properly with the make command. The make file should be written in such a way that it compiles only the files that are "changed" and files that are dependant on that changed file (and so forth).
4. **(1 Point)** Proper use of header files to provide as well as limit the exposure of functions and variables to other C files.
5. **(6 Points)** For displaying the polynomial. The polynomial should be printed in the increasing order of the exponents. You can incorporate the logic to store the terms in sorted order in the linked list as and when you are adding each term into the polynomial linked list.

```
$ cat data1.ssv
12 3
3 0
4 5
$ ./polyapp data1.ssv
0x2+12x3+4x5
for x=-2, y=-224
for x=-1, y=-16
for x=0, y=0
for x=1, y=16
for x=2, y=224
```

6. **(4 Points)** For merging polynomial terms with the same exponent. You will have to incorporate this logic when you are adding each term into the polynomial linked list one-by-one. (You may still display polynomial terms whose coefficient evaluates to 0 as such and is not necessary to remove them).

```
$ cat data2.ssv
12 3
3 0
-4 2
4 5
3 2
$ ./polyapp data2.ssv
3x0-1x2+12x3+4x5
for x=-2, y=-225
for x=-1, y=-14
for x=0, y=3
for x=1, y=18
for x=2, y=223
```

7. **(4 Points)** For correct evaluation of polynomial expression for different values.

All coefficients will be whole number integers (-negative through zero to positive). All exponents will be whole numbers (zero through positive integers).

** **Important** ** If your program "hangs" / is stuck while executing it through the tester script and requires TAs to interrupt it, you will loose **4 points**.

WHAT TO HAND IN

Turn in the make file `Makefile`, your git log `gitlog.txt`, C program source codes `polyapp.c`, `poly.c`, and `utils.c` named properly **AS WELL AS** any header files (`.h`) that your program needs to compile correctly. You do not have to zip the files. The files must be uploaded to mycourses. If you zip your files, double check to ensure that they are not accidentally corrupted. DO NOT turn in the executable `polyapp` or your testing data files. TAs will compile your C program on their own as indicated in the problem descriptions above.

MISC. INFORMATION

There is a tester script `mini6tester.sh` that is provided with the assignment that you can use to test how your program is behaving. TAs will be using the exact same tester script to grade your assignment.

```
$ ./mini6tester.sh
```

However, it is recommended that when you start writing your program, test it yourself first with the above examples. Make a test case for each scenario by creating your own data files that have the format mentioned above. Once you are fairly confident that your program is working, you can test it using the tester script.

You can compare the output produced by running the mini tester on your C program to that produced by the mini tester on the solution programs which is given in `mini6tester.out.txt`.

Please note that, the output of test cases 1 and 2 might be different based on the header files you have created. We have not included a sample of this test case as it will give away the solution part. The tester script will try to change the timestamp of every C and H files in your directory in an attempt to figure out how your make file works. Therefore, if you have any C or header files that you are not incuding for your assignment (such as backup files), I recommend moving it to a different directory (Even a sub-directory) before running the mini tester script to ensure it works the way it will when TAs run it. You should not receieve any erros and warnings from make when mini tester runs it if you have written a proper make file.