School of Computer Science, McGill University

COMP-206 Introduction to Software Systems, Winter 2020

Mini Assignment 5: C Programming - Functions and File I/O

Due Date March 13, 23:55

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. You can also reach out to the course email address, utilize TA/Instructors office hours as necessary. 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.

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`, or you can go to the third floor of Trottier and use any of those labs to ssh to `mimi` to complete this assignment. 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 one C program. 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 Banking System Application

In this exercise, you will create a C program, `bankapp.c` to implement a simple banking application.
Your application provides three capabilities: (i) Add an account number, (ii) Make a deposit, (iii) Make a withdrawal. All of your banking data is stored in a CSV file `bankdata.csv` which has the following format. You can use vi to create a sample data file of your own for testing purposes.

```
AC,2023,Jane Smith
TX,2023,2020-02-10,1023.34
AC,5023,Sally Long
TX,2023,2020-02-10,-103.34
TX,5023,2020-01-15,78.00
```

As can be seen above, there are two kinds of records in this CSV file. (i) Account records, indicated by `AC` in the first field, followed by a 4 digit account number, followed by the name of the account holder (max 30 characters long). (ii) Then there are transaction records that are indicated by `TX` in the first field, followed by the account number, transaction date in `YYYY-MM-DD` format, followed by the amount of money deposited or withdrawn (negative values for the latter).

1. Use vi to create a C program source file `bankapp.c`
2. **(1 Point)** The program 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. **(1 Point)** The source code should be compilable by (exactly) using the command `gcc -o bankapp bankapp.c`
4. **(1 Point)** The compilation step should not issue any warnings.
5. **(1 Point)** All the error messages must be printed to `stderr` and not `stdout`.
6. **(2 Points)** Your program's usage is as follows.
   To add a new account

```
$ ./bankapp -a ACCTNUM NAME
```

   To make a deposit

```
$ ./bankapp -d ACCTNUM DATE AMOUNT
```

   To make a withdrawl

```
$ ./bankapp -w ACCTNUM DATE AMOUNT
```

   Any other attempt must throw a usage error. **You may however, chose to ignore the extra arguments passed to any of the above valid options at your discretion and continue processing.** The exit code for invalid usage of your program should be 1.

```
$ ./bankapp
Error, incorrect usage!
-a ACCTNUM NAME
-d ACCTNUM DATE AMOUNT
-w ACCTNUM DATE AMOUNT
$ echo $?
1
```

   In case the user passed a valid option but did not pass sufficient arguments to process it, provide a more customized error message.

```
$ ./bankapp -a 1024
Error, incorrect usage!
-a ACCTNUM NAME
$ echo $?
1
```

```
$ ./bankapp -d 1024 2010-02-12
Error, incorrect usage!
-d ACCTNUM DATE AMOUNT
$ echo $?
1
```

```
$ ./bankapp -w 1024
Error, incorrect usage!
-w ACCTNUM DATE AMOUNT
$ echo $?
1
```

7. **(1 Point)** Your program should read the data contents from a CSV file, `bankdata.csv`. It must be present in the current directory. If the program cannot find it, it must thrown an error and terminate with code 100.

```
$ ./bankapp -a 1024 "John Smith"
Error, unable to locate the data file bankdata.csv
$ echo $?
100
```

8. **(2 Points)** Your program must allow new accounts to be added. These accounts will be appended to the data file `bankdata.csv`.

```
$ ./bankapp -a 1024 "John Doe"
$ tail -1 bankdata.csv
AC,1024,John Doe
```

If the account number already exists (as indicated by the `AC` records), then the program should not add it, instead throw an error message to indicate this and terminate with error code 50.

```
$ ./bankapp -a 1024 "John Doe"
Error, account number 1024 already exists
$ echo $?
50
```

9. **(3 Points)** Your program should facilitate deposits to an existing account. A transaction record indicating the amount of money to be deposited is then appended to the data file.

```
 $ ./bankapp -d 1024 2020-02-12 334.52
 $ tail -1 bankdata.csv
TX,1024,2020-02-12,334.52
```

If the account does not exist, it should throw an error message and terminate with code 50.

```
$ ./bankapp -d 1025 2020-02-12 334.52
Error, account number 1025 does not exists
$ echo $?
50
```

10. **(4 Points)** Your program should facilitate withdrawals from an existing account. A transaction record indicating the amount of money that is withdrawn is then appended to the data file.

```
$ ./bankapp -w 1024 2020-02-14 20.00
$ tail -1 bankdata.csv
TX,1024,2020-02-14,-20.00
```

If the account does not exist, it should throw an error message and terminate with code 50.

```
$ ./bankapp -w 1025 2020-02-14 20.00
Error, account number 1025 does not exists
$ echo $?
50
```

If the account does not have enough balance, it should display an error message, and also indicate the current balance in the account. Terminate with code 60. You should be able to compute the existing balance on the account by adding up all the deposits and withdrawals on the account. Make sure that the amounts you print on the screen has only 2 decimal places.

```
$ ./bankapp -w 1024 2020-02-15 2020.00
Error, account number 1024 has only 314.52
```

11. **(1 Points)** Remember to terminate the successful execution of your program with code 0.
12. **(2 Point)** Your program must not crash because the data file is empty (or at any other situation). Instead it should do the proper processing according to the scenarios already described earlier. (i.e, add a new account, let the user know the account does not exist for a transaction, etc.).
13. **(1 Point)** Your program should include some function definitions of your own. At a minimum, you should have one function per operator. You are free to have additional functions in your program.

** **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**. If your program causes corruption of the data file for any of the test cases, you will loose **4 points**.

WHAT TO HAND IN

Turn in the C program source code `bankapp.c` named properly. You do not have to zip the file. The file must be uploaded to mycourses. DO NOT turn in the executable `bankapp` or your testing CSV file `bankdata.csv`. TAs will compile your C program on their own as indicated in the problem descriptions above.

MISC. INFORMATION

There is a tester script `mini5tester.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.

```
$ ./mini5tester.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 `bankdata.csv` file that has 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 `mini5tester.out.txt`.

FOOD FOR THOUGHT!

The following discussion is meant to encourage you to search independently for creative and optimal ways to perform rudimentary tasks with less effort and/or make your program robust and sophisticated. It does not impact the points that you can achieve in the above questions.

- As we know, I/O is a slow process and can be costly if you have very large files. Can your program make decisions in certain scenarios (both fail cases and successful cases) without having to read all of the data file?