School of Computer Science, McGill University

COMP-206 Introduction to Software Systems, Fall 2021

Mini Assignment 5: C Programming

Due Date December 3, 2021, at 18:00 EST

GENERAL INSTRUCTIONS

PLEASE SEE AT THE END OF THIS DOCUMENT RULES ABOUT CHEATING AND PLAGIARISM.

This is an individual assignment. You need to solve these questions on your own. If you have questions, post them on ED, but do not post major parts of the assignment code. Though very small parts of code are acceptable, we do not want you sharing your solutions (or large parts of them) on ED. If your question cannot be answered without sharing significant amounts of code, please make a private question on ED or utilize TA/Instructors office hours. Late penalty is -20% per day, max 2 days late. After 2 days your assignment will not be graded.

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 access mimi.cs.mcgill.ca from your personal computer using **ssh** or **putty** as seen in class and in Lab A. All your solutions should be composed of commands that are executable in mimi.cs.mcgill.ca.

Questions in this assignment require you to submit bash scripts. Make sure to add your name and student ID as comments at the top of the scripts as proof that the work submitted is yours.

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**. (Commands that execute 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 lose up to 3 points for not following the instructions.

Labs H and I provide some background help for this mini assignment.

Total Points for this assignment: 20

This assignment has 1 exercise with 8 questions.

QUESTION INTRODUCTION

Bank website portals keep a log of all the transactions performed during a day. At the end of the day, the administrator will run a program that verifies that the bank records agree with the transaction log. This assignment asks you to write a bank transaction verification program.

The bank uses two log files. One log file, called `state.csv`, describes the state of all the bank customers accounts at the beginning of the day before transactions were made at the portal, and the state of all the accounts at the end of the day after all the transactions were processed. The second log file is called `transactions.csv` and it records all the transactions as input from the portal before they were applied to the account. Important: we cannot not know the number of records in these two csv files before the program begins.

The transaction verification program re-runs all the transactions on temporary bank accounts given the initial state and compares the final state of the temporary accounts with the final state of the real accounts. Everything is considered good when the end state of the temporary accounts agrees with the end state of the real accounts, otherwise error messages are displayed.

We ask you to write the verification program in the following way:

Ex. 1 — Bank Transaction Verification Program (Total 20 Points)

Goal: To make a C program that verifies that the bank account's end state matches the day transaction log.

Source filename:
`TransactionVerification.c`

Program syntax:
`./tv STATE TRANSACTIONS`

Legal usage examples:

- `./tv state.csv transaction.csv`
- `./tv ../state.csv /data/log/transaction.csv`

1. **(1 Point)** The program's compiled filename must be `tv` use the `gcc -o` option.
2. **(1 point)** The program's source filename must be `TransactionVerification.c`
3. **(1 point)** Add comments to the top of the file stating your name and student ID number. This will serve as proof that you wrote this program on your own.
4. **(1 point)** Command Line Argument Error Checking

   The program accepts two filenames as command line arguments. The first filename is the state of accounts. The second filename is the day's transactions. Your program assumes these two files already exist; however, your program does not know the names of these files and must receive them as command line arguments. Your program assumes these two files are properly formatted CSV files.

The program terminates when the user does not provide two arguments. Do not check the filename, just check the number of arguments.

If there is an error, terminate the program with `exit 1` and display the error message: "`Wrong number of arguments!`" and then display below the message the **program syntax** and the **legal usage examples** (as we did in the previous assignment). The user should then see the command line prompt.

5. **(1 point)** Checking that the filenames are valid.

   After the program has verified the arguments are correct, now verify that the two arguments are legal filenames. Even though we assume the contents of the files are CSV formatted, the filenames do not need to use the CSV file extension. This test is simply whether the two arguments provided are file names and whether they are readable. Hint: use fopen().

   As in question 4, display the following error message "`Unable to open filename %s`", where %s is the name of the invalid file, and then display the **program syntax** and **legal usage examples**. The program terminates with `exit 2`. The user should then see the command line prompt.

6. **(5 points)** Reading the state.csv file.

   Every day the number of bank accounts can change. Therefore, we never know ahead of time the number of bank accounts in the `state.csv` file. **This means we cannot use an array to store these records**. We will need to use a linked list of struct. However, we do know the syntax of the CSV data. The `state.csv` file contains the following n rows of information:

   ```
   accountNumber,startingBalance,endingBalance
   ```

   The field `accountNumber` is a unique 6-digit integer number identifying the account. For example: 125625 or 982354.

   The field `startingBalance` is a floating-point number representing the amount of money in the bank account at the beginning of the day.

   The field `endingBalance` is a floating-point number representing the amount of money in the bank account at the end of the day.

   The contents of the file `state.csv` will look something like this (it has a header row and no spaces):

   ```
   accountNumber,startingBalance,endingBalance
   123456,525.32,300.10
   146782,1000.35,2001.10
   176541,20125.00,10500.00
   ```

   This means we have three bank accounts. Bank account 123456 started the day with $525.32 and ended the day with $300.10. Account 146782 started with $1,000.35 and ended with $2,001.10. The last account is 176541 and it began the day with $20,125.00 and ended the day with $10,500.00.

   To read the `state.csv` file correctly please do the following:

First: create the structure

```
struct ACCOUNT {
    int accountNumber;
    double startingBalance;
    double endingBalance;
    struct ACCOUNT *next;
};
```

Second: read line-by-line the `state.csv` file. For each line `malloc()` an `ACCOUNT` node and add it to a linked-list sorted by increasing `accountNumber` order. The fields from the CSV file are saved in the node. Repeat this for ever record in the CSV file.

Third: if an `accountNumber` already exists in the linked-list, then the CSV file requirement of unique account numbers has been violated. Your program must continue executing but it must display the following single-line error message: "`Duplicate account number [account, start, end] : %d %f %f`". The duplicate field is not added to the linked-list. To be clear, the program keeps the first occurrence of the `accountNumber` but does not keep the subsequent occurrences of the `accountNumber` and displays the error message for all subsequent occurrences.

Fourth: close the `state.csv` file.

7. **(5 points)** Reading the `transaction.csv` file.

   Every day a new `transactions.csv` file is created for that day's transactions. We do not know ahead of time the number of transactions in this file. However, we do know its CSV format:

   ```
   accountNumber,mode,absoluteValueAmount
   ```

   The field `accountNumber` is a 6 digit integer number. It is not unique in the file, because each transaction results in a record. For example, when account 123456 deposits $10 and withdraws $15 in the same day, this results in two records. One record for the $10 deposit and another record for the $15 withdrawal.

   The field `mode` is a single character `'d'` or `'w'`. The character `'d'` represents deposit. The character `'w'` represents withdrawal.

   The field `absoluteValueAmount` is the amount of the transaction as a floating-point number >= 0.0.

   An example `transaction.csv` file could look like the following (it has a header row and no spaces):

   ```
   accountNumber,mode,absoluteValueAmount
   123456,d,125.00
   123456,w,100.00
   456321,d,20.00
   987456,w,5000.00
   ```

   The above CSV file means: account 123456 did two transaction during the day, a deposit of $125.00 and a withdrawal of $100.00. Account 456321 deposited $20.00, and account 987456 withdrew $5,000.00.

To read the `transaction.csv` file correctly please do the following:

First: read each record of the CSV file one at a time and parse the fields.
Second: locate the account number in the linked-list. If it is not there, continue executing ignoring the record, but printing this single-line error message: "`Account not found (account, mode, amount): %d %c %f`". Then go back to the first step.
Third: Assuming the record has been located, then using the `mode` either increase the node's `startingBalance` or decrease the node's `startingBalance`. Display an error message if the `startingBalance` goes below zero. Reset the `startingBalance` to zero and continue executing. The single-line error message is the following: "`Balance below zero error (account, mode, transaction, startingBalance before): %d %c %f %f`".
Fourth: go to the first step processing the next record in the `transaction.csv file`.
Fifth: When the `transaction.csv` file has been processed completely close the `transaction.csv` file.

8. **(4 points)** Final Statistics.

   After the program has processed all the transaction from the `transaction.csv` file, the program does a final scan of all the accounts in the link list comparing the `startingBalance` with the `endingBalance`. These two balances should now agree after we updated the `startingBalance` with all the transactions of the day.

   Display a single-line error message for every account where the `startingBalance` is not equal to the `endingBalance`. Display the following single-line error message: "`End of day balances do not agree (account, starting, ending): %d %f %f`".

Important things to consider:

- There may be accounts in `state.csv` that had no transactions in `transaction.csv`.
- It is possible that `state.csv` and/or `transaction.csv` may be empty. This is only a problem when `state.csv` is empty and `transaction.csv` is not empty. In this case the following single-line error message is displayed: "`File state.csv is empty. Unable to validate transaction.csv.`" The program exits early with error code 3.
- **(1 point)** Your program must deallocate memory before it ends (in all cases of ending).

Your program is now complete.

Important: FILENAME must support absolute and relative paths.

Turn in `TransactionVerification.c`.

THE TESTING SCRIPT

To use the script **mini5tester.sh**, copy it into the directory where your **tv** executable is located. The script **mini5tester.sh** has no arguments. For example, to run the tester you will type:

./mini5tester.sh

If the script is in the same directory as **tv** (and your executable is called **tv**) then it should work. This testing script does not output any of its own error messages. You will simply see the output from your program.

I suggest you **more** or **vi mini5tester.sh** to see what it does. In general, the script invokes your script in a way that should result in a correct execution, and then it invokes your script in a way that should result in an error. The testing script has comments for each line it thinks should work and fail. You can use these expected outcomes to help fix your program.

The teaching assistants will grade with a **modified testing script** having additional testing cases that will look at boundary cases. Make sure to handle boundary cases.

WHAT TO HAND IN

In the assignment box on myCourses for mini 5, submit the `TransactionVerification.c` source file only.

The TA will compile your source file on mimi to confirm that it was created there. If there were any error when compiling the assignment will be graded with 0.

Your program should not display any unintentional error messages. Your program MUST run on mimi to be graded. The TA will not edit your assignment to get it running. The TA will grade your assignment using the testing script provided. The TA may add extra testing options to the script they will use.

FOOD FOR THOUGHT! or FOR THE GLORY!

For those students who felt the assignment question was too easy or wanted a further challenge, each assignment will have an extra problem called either Food For Thought! or For The Glory!. This extra problem is optional. The TA will not grade it. This is for you to explore. It is for you to brag to your friends about.

Food For Thought! problems are discussion and exploration questions. For The Glory! will be programming or problem solving related.

Food For Thought! Question:

> Which C library functions could you use to simplify the parsing of the CSV file? In other words, are there any library functions that can automatically parse the CSV file in the gcc installation we have on mimi?

DO NOT HAND IN THIS FOOD-FOR-THOUGHT! QUESTION (But I would be interested in hearing what you did! Email it to your prof. – but only if it is completed)

Rules about Cheating and Plagiarism

McGill takes seriously cheating and plagiarism. To make it clear to the student what is permitted and not permitted the following checklist has been included. **The rows that are highlighted in yellow are permitted**. The rows that are not highlighted are NOT permitted.

NEXT PAGE: Student Code of Conduct Checklist