School of Computer Science, McGill University

COMP-206 Introduction to Software Systems, Winter 2024

Assignment 3: Bash CSV analysis and mini C calculator

Due Date March 12, 17:59 EST
TA Office Hours will start on Feb 28: See MyCourses announcement for list

Before you start, please read the following instructions:

- **Individual assignment**
  This is an individual assignment. Collaboration with your peers is permitted (and encouraged!) provided that no code is shared; discuss the ideas.
- **Use Discord and OH for questions**
  If you have questions, check if they were answered in the #clarifications channel on Discord or in the Discord q-and-a forum. If there is no answer, post your question on the forum. Do not post your code. If your question cannot be answered without sharing significant amounts of code, please use the TA/Instructors office hours. **Do not email the TAs and Instructors with assignment questions.** TAs should only be contacted by emails if you have questions about a grade you received.
- **Late submission policy**
  Late penalty is -10% per day. Maximum of 2 late days are allowed.
- **Must be completed on mimi server**
  You must use `mimi.cs.mcgill.ca` to create the solution to this assignment. An important objective of the course is to make students practice working completely on a remote system. You cannot use your Mac/Windows/Linux terminal/command-line prompt directly without accessing the mimi server. You can access `mimi.cs.mcgill.ca` from your personal computer using **ssh** or **putty** as seen in class and in Lab A. **If we find evidence that you have been doing parts of your assignment work elsewhere than the mimi server, you might lose all of the assignment points.** All of your solutions should be composed of commands that are executable in `mimi.cs.mcgill.ca`.
- **Must be completed with vim**
  A goal of the class is to get you to be comfortable in a command-line text editor. This wouldn't happen just from knowing the vim commands from slides: You become comfortable by using it a lot. Assignments are a good opportunity for that.
- **You must use commands from class**
  You may not use commands that you haven't seen in class. If you are not sure, ask on the Discord server.
- **Your code must run**
  For this assignment, you will have to turn in three bash programs and 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. Your scripts should not produce any messages/errors in the output unless you explicitly produce it using an echo statement. Your script should run near-instantly: Hanging is usually a result of some logical error in your code. Your scripts must not create any files internally, even as temporary output storage that the script deletes by itself. DO NOT Edit/Save files outside of mimi, not even to edit comments in the scripts. This can interfere with the file format and it might not run on mimi when TAs try to execute them. **TAs WILL NOT modify your scripts in any ways to make them work.**
- **Please read through the entire assignment before you start working on it**
  You can lose up to 3 points for not following the instructions in addition to the points lost per questions.

Total Points: 20

Ex. 1 — Bash CSV analysis (5 points)

Background information

More and more, large datasets are made available to the public. Cities publish information about each intervention by their emergency services, meteorological organizations publish hourly reports from each of their weather stations, scientists publish results from their experiments, etc. In this exercise, you will see how you can use the Bash commands you've learned in class to analyse these datasets.

The most popular file format to share large datasets are CSV files. A CSV file is a text file formatted on specific rules used to represent a dataset. Each data point (record) is one row in the text file (ie. Starts at the first character and terminates with a newline). Each data point information element (field) is separated by a comma. Therefore, the comma and the newline are reserved symbols for this format and cannot be used inside of fields. Often, the information contained in the first line is the description of each field: This is called the header. Here is an example CSV file:

```
name,age,money
Mary,10,100.00
Sahib,15,200.00
```

This is a CSV file that contains a header followed by two records. Each record contains three fields. In this example the first field is the name of a person. The second field is that person's age. The last field is the amount of money they have.

Analyzing the World Cities dataset

In this exercise, you are provided with the CSV file `worldcities.csv`[1]. This CSV file contains the population for 44691 prominent cities (large cities, capitals, etc.). Open the file (using vim or cat etc.) to see the header. Each record contains the name of the city, its latitude and longitude, its country, the country iso codes, whether the city is a capital and at which level, and the population. For this exercise, we will focus on the city, country, and population fields.

Create a bash script named `analyse`. Do not forget the shebang. This script will handle three different analysis tasks. As in the previous assignment, it will know which task to do based on the first command line argument it receives, i.e. the usage is `./analyze <task> <additional_arguments>`. The tasks are: `avg`, `total`, `diff`. If the first command line argument is not one of these, print an appropriate error message and exit with error code 1. You should then implement each task as follows (Hint: The `awk` command will be very useful! There are examples of using `awk` with CSV files in the slides, and you can also look online for more examples):

1. **The `avg` task.**

   In this task, the user will input a country, and the script will return the average population for the cities listed for that country in the dataset. You can keep as many decimals as you wish (or use integer division).

   The syntax for this task is `./analyse avg country`

   An example is:

   ```
   ./analyse avg "Canada"
   85371
   ```

   If there isn't exactly one additional argument, exit with error code 2. You do not need to handle what happens if the country isn't in the dataset, but make sure that it does work for any country that is in it.

[1]CSV file taken from simplemaps under the CC BY 4.0 license. Some columns were removed.

2. **The `total` task.**

   In this task, the user will input a country, and the script will return the total population for the cities listed for that country in the dataset.

   The syntax for this task is `./analyse total country`

   An example is:

   ```
   ./analyse total "United States"
   380148373
   ```

   If there isn't exactly one additional argument, exit with error code 3. Once again, you do not need to handle what happens if the country isn't in the dataset, but make sure that it does work for any country that is in it.

3. **The `diff` task.**

   In this task, the user will input two cities, and the script will return the difference in population between the two cities. For example, if the first city has a population of 18972871 and the second city has a population of 3519595, then the difference is 15453276. It is fine if there is a negative sign.

   The syntax for this task is `./analyse diff city1 city2`

   An example is:

   ```
   ./analyse diff "New York" "Montreal"
   15453276
   ```

   If there isn't exactly two additional arguments, exit with error code 4. You do not need to handle what happens if one of the cities isn't in the dataset, and you can ignore cases where more than one city has the same name.

Ex. 2 — A mini calculator (15 points)

This problem has two parts. The first is to write a C program capable of performing various operations according to its command-line arguments. The second is to write two bash scripts for facilitating the development of the C program. You should work on these two problems at the same time, since your code for part 2 will help you to develop your code for part 1.

Learning goals assessed:

1. Write robust system interactions in C.
2. Use appropriate data structures and algorithms in C.
3. Organize code in C for readability.
4. Write bash scripts for automating development tasks.

Learning goal not assessed: look up and use documentation during development.

For this exercise, grading will be done using a rubric based on these learning goals. You can see the rubric in the "Rubric for mini C calculator" section below. You can use the rubric to help you know what to improve when working on your assignment.

The mini calculator itself

In this exercise you will write `minicalc.c`, compiled as `minicalc`.

This program's input is its command-line arguments. The command-line arguments specify an operation to perform together with its operands. The usage of the mini calculator is `./minicalc OP ARGS...` where the number of arguments will depend on the chosen operation.

The required operations are the following. In the below, `N1`, `N2`, etc stand for user-supplied ints; `A1`, `A2`, etc stand for user-supplied doubles; and `S1`, `S2`, etc stand for arbitrary strings. In all cases, your program should produce its expected output followed by a newline character. In case of success, your program must exit with code 0.

1. Addition: `./minicalc + N1 N2`
   Output: the sum of the integers.
2. Greatest common divisor:[2] `./minicalc gcd N...`
   Arbitrarily-many integer arguments may be supplied and your program should output the GCD of them all. Implement this using a loop. You can take code from online to find the GCD of two integers, just cite it in your solution. Your code must include a comment explaining at a high-level why the algorithm works.
   Output: the GCD of all the given integers.
3. Square root: `./minicalc sqrt A1`
   Output: the square root of the number. Calculate this using the standard library function `sqrt`.
4. Anagram: `./minicalc anagram S1 S2`
   Output: `true` if the strings `S1` and `S2` are anagrams, else `false`.

Anagrams

One word is an anagram of another word when its letters can be rearranged to form the other word. For example, "brainy" and "binary" are anagrams.

Robustness of the mini calculator

Your minicalc implementation must be robust in the face of possible user input: it must validate user input properly and use safe functions only. Recall the in-class demonstration of the unsafeness of `gets`! Your program should take care to avoid potential buffer overruns.

Your implementation must gracefully handle the following failure scenarios, as specified. In case of failure, it must print an appropriate error message (of your choosing) to stderr and **exit with the specified code.**

- Minicalc requires at least one command-line argument.
  Exit with code 1 otherwise.
- The first command-line argument must specify one of the allowed operations.
  Exit with code 2 otherwise.
- According to the selected operation, the number of subsequent command-line arguments must be as required. For addition and anagram, it's exactly two; for square root it's exactly one; for gcd it is at least two.
  Exit with code 3 otherwise.
- According to the selected operation, the operands must be of the correct type.
  Use the C standard library functions `strtol` and `strtod` to convert strings into integers and doubles respectively, while checking whether the conversion is successful[3].
  Exit with code 4 otherwise.
- The input of the square root operation must be nonnegative.
  Exit with code 5 otherwise.
- The inputs of the anagram operation must consist only of lowercase letters.
  Exit with code 6 otherwise.
- The inputs of the GCD operation must all be greater than zero.
  Exit with code 7 otherwise.

Development automation for the mini calculator

Motivation. During the development of software in compiled languages such as C, the developer must run the compiler. As seen in class, this can require several flags and options. Typing out these commands every time we wish to build our application would be redundant and time-consuming. Moreover, if we wish to collaborate with

[2] `https://en.wikipedia.org/wiki/Greatest_common_divisor`
[3] See `man strtol` and `man strtod` for details.

another developer to write our code, we must agree on how it is to be compiled. Therefore, it becomes instrumental to create a script that automates compilation.[4]

Similarly, once the application is compiled, it must be tested. This can of course be done manually by running the application with some sample inputs – e.g. `./minicalc sqrt -4` to observe that an error message is printed and that the exit code is as required – but automating this task makes testing both faster and more consistent!

1. Write a bash script named `build` such that running `./build` compiles `minicalc.c` into an executable `minicalc`.
2. Write a bash script named `test` that automates testing `minicalc`. That is, running `./test` will execute `minicalc` in the same directory on a variety of inputs and check that the output and exit code are as expected.
    - Your test suite must be complete! Each failure mode of each operation must be tested.
    - Successful execution must also be tested. Capture the output of `minicalc` to compare it against a reference output. For example, your script could check that `./minicalc + 2 2` outputs `4` and exits with status zero.

Rubric for mini C calculator

The notation "LG" followed by a number refers to an assessed learning goal from the previous section. Each learning goal is associated in this table with the weight it holds in the overall assessment of your learning in this assignment.

If your code does not compile or run at all, it will be given a grade of zero!

The categories A, B, C in the below table do not necessarily match the standard interpretations of those letters in McGill's grading scheme. Instead, they should be understood as "exceeds expectations," (A) "meets expectations," (B) "does not meet expectations," (C) and "not assessable" (N).

	LG1 (60%)	LG2 (10%)	LG3 (10%)	LG4 (20%)
A	All failure scenarios are handled and care is taken to avoid buffer overruns. Input validation logic is clearly separated from business logic.	Loop conditions clearly reflect the structure of the data. Array offset calculations for indexing are straightforward. Algorithms handle all edge cases properly.	Code is well signposted with comments that explain the why, not the how. Code is decomposed into functions each with a clear, unambiguous purpose. Code redundancy is minimal. Nesting of control flow structures is generally limited to two or three deep. Code is well-formatted with proper indentation and judicious use of blank lines to improve readability.	Automation scripts are concise. Bash functions are used to minimize redundancy. The test suite is exhaustive, covering all failure scenarios and one success scenario per operation.

[4] In practice for large developments, simple bash scripts are not used for this, but rather more sophisticated build systems such as Makefiles are. For our purposes in this assignment, a bash script is more than sufficient. You will learn about Makefiles later in the course.

B	Most (˜70% of) failure scenarios are handled. Buffer overruns are possible but unlikely. Input validation logic and business logic are largely separate, but sometimes intermixed.	Most loops are expressed with clear conditions. Array offset calculations are at times hard to follow. Algorithms leave some edge cases unhandled.	Code comments explain the how, but not the why. Code is decomposed into functions, but their purpose is sometimes unclear or ambiguous. Code is redundant, but not excessively. Control flow nesting is deep.	Automation scripts are functional, but redundant. The test suite covers most failure scenarios. One success scenario per operation is tested.
C	Failure scenarios are generally not handled. Error messages do not necessarily reflect the problem, and the used exit codes largely do not match the specification. Input validation logic is haphazard and generally intermixed with business logic. The handling of buffers is improper and overruns are easy to arrange by the user.	Loop conditions and control flow are confusing or unclear. Array offset calculations are overall confusing. Algorithms do not handle edge cases.	Code comments add no value or are largely missing. The use of functions is limited, or the present functions have unclear purposes. Code is very redundant, with significant pieces appearing copy-pasted.	The scripts are functional but redundant. The test suite covers a few failure scenarios. Some success checks are missing.
N	If the program runs, failure scenarios are not checked. Error messages are not generated. The program generally assumes correctness of inputs. The program is easily made to crash or overrun buffers by the user, even on valid inputs.	The use of loops and arrays is generally incorrect. Loop conditions are outright wrong. Indexing logic is convoluted. Algorithms do not handle edge cases and produce incorrect results on typical inputs, too.	The code lacks comments. It is not decomposed into functions. Code is very redundant and highly nested.	The build script does not build `minicalc` or the test script does not actually run `minicalc`. If the tester does run, it checks a few cases at most. Test cases are invalid.

WHAT TO HAND IN

Upload your four files, `analyze`, `minicalc.c`, `build`, `test` to MyCourses under the **Assignment 3** folder. You do not have to zip the files, but you might need to if MyCourses does not accept the file format. Re-submissions are allowed, but note that TAs will only grade the most recent submission. **You are responsible to ensure that you have uploaded the correct submission.**