CSE 30 Spring 2022 Programming Assignment #2 - (Vers 1.2) Due Saturday April 23, 2022 @ 11:59PM

Remember: 'Start Early, Start Often'

Assignment – Data Extraction, Pointer Practice and Modifying Code

The goals of this PA are:

1. To get practice writing code using C pointers and strings
2. Learn about Delimiter-Separated-Values (also known as DSV) data format (a generalized version of Comma-Separated-Values or CSV files).
3. Get familiar with the New York City Traffic Ticket data (in CSV format) that we use with the in-memory database assignment in PA3
4. Practice with C library routines for reading lines of text, parsing command lines options and converting strings to integer and unsigned long values
5. Write routines that we will reuse in PA3 to read from a CSV file and load data into the PA3 database (a data handling process called **ETL** - **E**xtract **T**ransform and **L**oad)
6. **Learn to modify code you did not write** (something you will do a lot of when you graduate). Practice figuring out someone else's code and "fixing or finishing it".

A common format for moving data between systems is called a **DSV** file. A ***DSV file*** stores tabular data (numbers and text) in plain ASCII text. A **delimiter-separated values** (**DSV**) file uses a single character delimiter (typically `','` OR `'#'` OR `' '` OR `'\t'`, `etc`) to separate values on a line of data. A ***CSV file*** is a DSV file where the delimiter is a comma.

Each line of the file, or ***"data record"***, is terminated by a newline (`'\n'`). Each record consists of one or more data fields (or columns) separated by the delimiter only ***between*** the fields. Or stated differently, a delimiter is found after each field except for the last field in a record, which is always followed by a newline. In a proper DSV file, each record (line) will always have the same number of fields (columns). Typically the first record in a DSV file (the header record) is not part of the data but is documentation of the names (meaning) of each of the columns.

Fields are numbered from left to right starting with column 1 (or field 1). Fields can be empty. For example, a record that looks like `"A,,C"` has three fields where field 1 is **A**, field 2 is empty, and field 3 is **C**. Records are numbered starting from 1 (the first record in the file).

In terms of C, an entire record is processed as a single string. As we have discussed in lecture, you can use pointers to break a string into tokens or fields in place, without incurring the cost of copying the string. Each token will be a string. An empty record (one where there are no characters between the adjacent delimiters) is a string where the only character is the string termination character, a '\0'. The technique to break a string into tokens, shown in class, will be at the heart of this assignment.

In this programming assignment you will build a tool to extract data from DSV formatted data read from stdin (with different characters for delimiters), reformat it by dropping or moving fields within the record and writing it out to **stdout**. You will reuse the extraction routines from this PA to load the Data From the New York City Parking tickets CSV file into the in-memory database you will build in PA3. In the field of data analytics the process of reading data, modifying the data and then loading the data into a database is called Extract Transform Load (or **ETL** for short).

The NYC data file is in **CSV** format and consists of about 1.146 million records where each record contains 43 fields of data, and the total file size is 2GB. In PA3 you will **use only 5 fields** in each record, so **you must** eliminate the other unused fields out of the 43 fields. When working with datasets at this size, program runtime performance is an important consideration in the design and implementation of your solution.

To make testing easier, command line options to your program will allow a user to process only a subset of the records. This is accomplished by specifying record ranges (by line number) to extract.

Below we show an extracted subset of that file, with a reduced field count, of the header record followed by four data records.

```
Summons Number,Plate ID,Registration State,Issue Date,Violation Code
1105232165,GLS6001,NY,07/03/2018,14
1121274900,HXM7361,NY,06/28/2018,46
1130964875,GTR7949,NY,06/08/2018,24
1130964887,HH1842,NC,06/07/2018,24
```

In the above, there are five fields extracted from the 43 fields in the input data. The name of the fields is Summons Number, Plate ID (the license plate number), Registration State (the state that issued the plate), Issue Date (the date of the ticket), Violation Code (the type of the ticket – encoded by a number from 1 to 99). In PA3 you will use this extracted data to build an in-memory database where you will make queries by license plate number to get the tickets and total fines outstanding for that vehicle (and update the database simulating when fines are paid).

DSV Format Specifications

In this programming assignment (PA2), you will write a program called extract that will process DSV files and conform to the specifications numbered below. These specifications are a modification (simplification) of rfc4180 *"standard"* (https://www.ietf.org/rfc/rfc4180.txt) for CSV files.

In the specifications given below, we use a single comma as the field separator on the input file. In your solution, this would be any input delimiter as specified with command line options (, is the default).

Also note, that in the specifications, a descending "$_b$" is a single blank and a descending "$_n$ " is a single newline.

1. Each and every record is located on a separate line and terminated by a newline $_n$.
2. For this assignment, you can assume that every record is properly terminated with a newline $_n$.
3. All records in the file contain exactly the same number of fields.
4. Delimiters are only placed between fields. The number of delimiters in a record is one less than the number of fields. The last field in each record is only (and always) followed by a newline.

   Here is a comma delimiter

   aaa,MMM,ccc$_n$

   and the same record contents but using a colon ‘:’ delimiter

   aaa:MMM:ccc$_n$

5. The number and position of blanks in a record must be preserved. Below are some examples of normal record fields (also sometimes called columns) and the last two show records where there is one empty field. Obviously, a field cannot contain a newline $_n$

   aaa,MMM,ccc$_n$

   zzz,123,ccc$_n$

   zz$_{bb}$z,X$_{bbbb}$Z,$_b$ccc$_{bn}$

   zzz,123,$_n$

   aa,,cc$_n$

6. There may be an optional header line appearing only as the first line of the file with the same format as a normal line. The header will contain names corresponding to the fields in a record. The header record has the same number of fields as the rest of the data records in the file. Header records are treated like any other record. As an example here is a two record file, a header record and one data record:

   ```
   Summons Number,Plate ID,Registration State,Issue Date,Violation Code
   1105232165,GLS6001,NY,07/03/2018,14
   ```

7. Within a record there are two types of fields: **normal fields** and **quoted fields**. All fields in the record can be classified as either a normal field or a quoted field. Normal fields and quoted fields can be mixed within a single record. In a **quoted field, the first and last character in the field must be double quotes "**. In a **normal field, double quotes " cannot occur anywhere within the field**. If you see double quotes

" in a normal field, that record has an error in it. A **quoted field must have all the characters in the field between the quotes.** If the first field in a record is quoted, the first character in the record is a double quote ". Quoted fields always end with a double quote " immediately followed by either a delimiter or a newline when it is the last field in a record.

`"aaa","MM`$_b$`M","ccc"`$_n$

`"zzz",123,ccc`$_n$

The following is an example that contains incorrectly quoted fields:

`"aaa"c,"MM`$_b$`M,b"ccc"`$_n$

8. A quoted field can contain any character including the delimiter (**Exception: for this PA a newline cannot be inside a field**) with one rule for double quotes. If a double quote is to be found in a quoted field, it must appear as a **pair of double quotes ""** (two double quotes **""** immediately next to each other). No isolated/unaccompanied double quote " can appear in a quoted field (that field would have an error). Here are some examples of using "" and , (the comma delimiter) inside a quoted field. There are two fields in the first record and there are three fields in the second record. The fields are highlighted. As a note, the number of double quotes **" in a properly formatted field is always an even number (**two for the start and end, plus zero or more pairs of double quotes).

   `"Aaa""Bbb","x,yz"`

   `"a,ha","""the Great one"",""Rules,""Now","ccc"`$_n$

9. A record with just one field is also possible. It will have just the field and no delimiters in the record. A special case is with a single field record, you cannot have a record with just an empty field (no lines with just a newline in the file).

Usage

The program that you write will operate like a Linux command line utility.

1. It will read the input records from the standard input stream, one line (one record) at a time
2. Process each input record as specified above to create an output record
3. Write the output record to standard output stream one line (one record) at a time
4. Write error messages to the standard error stream (if any).
5. If there are no errors in the input stream, the program will return EXIT_SUCCESS from main().
6. If there are errors in either the input stream or while processing the command line options, the program will return EXIT_FAILURE from main().

Specifically, your program (which is called extract) will perform the following steps in a loop until EOF (end of file) is reached on the input stream:

1) Read one record (one \n terminate line) at a time using the library function getline() (see man 3 getline) from the standard input stream.
2) Determine if the record lies within the ***optional specified record number range***. This is changed through a command line option. The default is to process all records. As a result of the record number check, the program will either process the record, skip the record, or exit the program. See the section below that describes the command line arguments -b and -e on how the record number range limits work.
3) Break the record into null terminated tokens, using the specified input record delimiter, one token per column/field. Each token is pointed at by an array of pointers, one pointer per input record field/column.
4) Check that all fields/columns are valid per the specification above. Make sure that each record has exactly the specified number of fields/columns. Remember you can assume every record ends in a newline.
5) If the record is not valid, skip processing that record and write to stderr an error message using the supplied routine dropmsg() that is located in the file misc.c.
6) If the record is valid, output the selected columns in the order they appear on the command line using the specified output delimiter (see the description of the command line below) to the standard output stream using the library function printf().

A typical execution of your program from the shell might look like this (using an input file with three (3) fields, output field 1 followed by field 3 using a comma (default) as the field delimiters on both the input and output file.

```
./extract -c3 1 3 < input > output 2> ERRORS
```

Description of the Command Line Options

```
./extract -c count [-s start] [-e end] [-i in_delim] [-o out_delim] col [...]
```

Flag	Description
-c	Specifies the number of fields (columns) in each record read from the standard input stream. This option is **mandatory**.
-s	Specifies the first/start record number to process on the input stream. The first/start record number must be less than or equal to the end record number (unless the end line number is 0. See the -e description for what 0 means) **Default: first/start record to process is record 1 (first line)**
-e	Specifies the last record number to process on the standard input stream. If after processing the input until EOF, you find that the input has less records than was specified with the -e option, then write to the standard error stream (use fprintf()) indicating that there were less records in the input than specified and exit with EXIT_FAILURE **Default: last record number is 0 (this means process all records until EOF)**
-i	Specifies the delimiter on the records read from the standard input stream. Be aware that certain

	delimiter characters may have a special meaning to the command line shell, so you may have to enclose them in quotes. See man bash for more details. **Default: input delimiter is a single comma: ,**
`-o`	Specifies the delimiter on the output records written to the standard output stream. Be aware that certain delimiter characters may have a special meaning to the command line shell, so you may have to enclose them in quotes. See man bash for more details. **Default: output delimiter is a single comma: ,**

`col`	Specifies one or more column (or field) numbers in each input record. Each column is separated by white space on the command line. The order of column numbers in the argument list specifies the column order in an output record (the output column list). At least one column must be specified to output. The column numbers correspond to the field/column numbers in an input record. Starting with field 1 on the left. The col number must be between 1 and the number of columns specified on the input with the -c option. Col numbers can be duplicated. Not all columns in a record need to be selected.
	Examples:
	`./extract -c 5 1 2 3     // select columns 1 2 3 and write in that order`
	`./extract -c 5 2 1 1     // select columns 1 2 and write in the order 2 1 1`
	`./extract -c 5 2         // select column 2 and write column 2`

If there are no dropped records or other errors, extract will return EXIT_SUCCESS, otherwise it will return EXIT_FAILURE.

Coding and Program Development Requirements

1. You will write your code only in C and it must run in a Linux environment on the pi-cluster.
2. **You will write your program only using pointer notation**. There **must not be any array notation** (i.e. [ ] characters) **in your code**. **IMPORTANT: Using array notation [] in your code you will result in a *zero* (0) on the assignment.**
3. The program part of the assignment will consist of two (2) files, extract.c, subs.c The contents of these files are described below.
4. You will use the supplied Makefile to compile your program.
5. You will use the two supplied routines in misc.c (you should not edit misc.c or misc.h).
6. You must test your program on the pi cluster.
7. You must ***first*** use the supplied test files (**subset of the MVP tests**) to develop your code.
8. You must pass all supplied MVP tests ***before submitting*** to gradescope for the complete set of MVP tests and the Field Trial Tests.

9. You should use the type ***unsigned long*** with variables that refer to record counts (record or line numbers processed or dropped) or the number of records in the input.

10. You may **only use** the following library functions**. IMPORTANT: The use of any other library functions in your code will result in a *zero* (0) on the assignment**:

```
    strtoul(), strtol(), malloc(), free(), fprintf() printf(), getline(),
getopt() (getopt() is used in the supplied code found in the file misc.c)
```

How to Approach This Programming Assignment

Step 1: Examine the supplied files in the Starter code

Starter C code can be found in misc.c (which you should not have to change as it is complete), extract.h, and extract.c which contains most of main() except for some missing parts.

Look over extract.c, this is where main() is located. The starter code has a lot of comments to help you organize the solution. Much of main() is done for you so you can focus on writing the routines in subs.c. There are parts of main that have been commented out, fix them. You are free to change code, change variable names, add variables or remove variables in main() if you want. Study how main() works so you can see what needs to be done in your code. Be aware that you will need to write almost all of main() in PA3.

Look over the contents of the file misc.c. In this file are the routines do_opts() and dropmsg(). do_opts() does all the command line option argument processing for you using the C library routine getopt(). You will have to write your own routine to use getopt() in PA3 so now is a good time to learn by example how getopt() works (see man 3 getopt as well). It is important to note that do_opts() does not process the list of specified output columns in the command line. It returns the argv index of the first output column specified on the command line (which you use in main() to process the output column list and set up the index array). It works this way so you can figure out the size of the index array in main() so you can malloc() the correct size. dropmsg() is the function you call when you drop a record that has an error and you need to print this action to stderr. Study the arguments to dropmsg() so you know how to use it.

In the file subs.c is where you will be writing most of your code. There are two functions in that file, split_input() and wr_row(). The bodies of these files have been removed and you must write them. **Be aware that you will be reusing split_input() in PA3 so make sure your code works well.**

Step 2: Fix main()

The main() routine is mostly finished. Some sections are missing so you will need to finish or fix them. The comments will tell you what to do. The goal is to study the overall program flow in main so you know what is expected to happen in the two routines that main() calls (plus any helper files) that you will write (in subs.c).

Step 3: Finish the two functions in subs.c with any helper functions you need

1. If you drop an input record, for any reason, use dropmsg() found in the file misc.c to write out which record number was dropped and why. **Do not spend too much time on this as the contents of stderr will not be graded (it is there to help you debug your program and help the user use the program).**
2. Be aware that dropmsg() uses argv to print the name of the program the error is associated with. This is good practice for all command line programs.
3. Remember that variables in your program that work with record numbers should be of type unsigned long.
4. Read the description headers for each function in subs.c. In the main body you will get hints on how to process the code and the calls to use to dropmsg() when an error occurs.
5. **If you define any helper functions in subs.c**, **define them as private to the file subs.c**. Do this by **defining them with the keyword static in front of the function type**. Example: `static int helper(void);` **Place private helper function prototypes at the top of subs.c below the last #include (there is a comment in subs.c on where to do this). Do not edit subs.h**

Step 3.1 Processing the input: split_input()

Walk the input buffer (parameter `char *buf`) using pointers only, looking for the delimiter or a '\n', and making sure to stop at '\0'. For each delimiter (parameter char delim) or the final newline '\n' after the last record, replace the delimiter or '\n' with a '\0' to terminate that token. Store the pointer to the next char in the next input array element (parameter `char **table`). Repeat this process until you either reach the end of the input line or fill all array entries. You should not need to zero out the table array on each pass, you just keep track of which element pointer is being processed. At the end of processing the input line, you should have the table array filled with pointers to the start of each field (column) in the record and each field is a properly '\0' terminated string. At the end of the loop you must check you have exactly the number of columns as specified on the command line in the record (in parameter `cnt`).

Error messages:

too many columns in a record call `dropmsg("too many columns", lineno, argv);`
too few columns in a record call `dropmsg("too few columns", lineno, argv);`

In the process above you have to find and process each field/column. To process a field/column (check if it meets the specification for a field and locate where it ends), you first check if the first char in a field is a " or not. If it is a " , then process this as a start of a quoted field, otherwise process it as the start of an unquoted field. Follow the rules for proper fields/columns described in a prior section.

Error messages for a " inside a unquoted field

```
dropmsg("A \" is not allowed inside unquoted field", lineno, argv);
```

Error messages for quoted fields

Quoted field where the last character in the field was not a quote.

```
dropmsg("Quoted field not terminated properly", lineno, argv);
```

Quoted field where the closing " was missing.

```
dropmsg("Quoted field missing final quote", lineno, argv);
```

Step 3.2 Outputting the New Records: wr_row()

Walk down the output array, (parameter `int *out_tab`), from the first element in the array. Each element in the output array contains the index number of the next field to write to the standard output stream. Use this number as an index into the input array, (parameter `char **in_tab`), to get the pointer to the input field you need to output (remember array indexes start at 0 and the columns are numbered from 1). The number of output record fields (also the number of elements in in_tab) is in the parameter `int out_cnt.` Use the parameter `char delim` as the delimiter in your output records that you write to the standard output stream.

There is one special case where wr_row() will drop a record on output. If there is only one output column and input record is an empty line, skip it (do not print it) and do the following

```
        dropmsg("empty column", lineno, argv);
        return 1 to main
```

If wr_row() does not drop a record, it returns 0 (the number of records dropped).

Step 4: Compile your program with Make

The Makefile will compile your program and create an executable called extract.
You are required to write code that compiles cleanly (using the compile options already specified in the Makefile – do not alter the Makefile or over-ride the options to gcc).

Step 5: Test your program and debug

The starter files contain a rudimentary test harness (software used to test the operational correctness of your program) to run some of the Minimum Viable Product Tests (public tests) on your program (see the section below on testing). In this PA, the test files are less comprehensive than in PA1, so to get through the MVP tests on gradescope you should add tests to the harness. **You must use the test harness to test your program before making your first submission to gradescope**. Feel free to add any additional tests you feel necessary to pass the tests that gradescope will use.

The test harness can be invoked from the Makefile. You can use the make target ***test*** to run the public tests on your program. This target will compile your program if required and then run all the tests.

```
% make test
```

Testing your software is just as important as developing it. Invoking the test harness has its own target in the Makefile so it is easy to run tests after you change your code. This is more important for software that is being maintained (after it has shipped) as simple code changes can break things (called a regression) and you want to run the test suite (the set of tests) to make sure the code still is fully functional.

Step 6: Comment and check your code follows the CSE30_C_Style document

To get all the style points, read over the style document and clean up your code by following the guidelines. In PA2 style will be more enforced than in PA1

Step 7 : Submit to Gradescope under the assignment titled "A2: extract"

After you pass all the MVP tests in the test harness and only then, will you submit to gradescope the following files: `extract.c subs.c`

You can upload multiple files to Gradescope by holding CTRL (⌘ on a Mac) while you are clicking the files. You can also hold SHIFT to select all files between a start point and an endpoint. Alternatively, you can place all files in a folder and upload the folder to the assignment. Gradescope will upload all files in the folder. You can also zip all the files and upload the .zip to the assignment. Ensure that the files you submit are not in a nested folder.

After submitting, the autograder will
1. Check all files were submitted
2. That the program compiles without errors
3. Rerun the public tests and then the private tests

Make sure to check the autograder's output while submitting!

Grading

The assignment will be based on 60 points:

5 points on programming style and comments

35 points on passing the MVP public tests **(stdout data only; stderr output will not be checked)**

20 points on passing the Field Trial private tests

Note: The coding style is expected to be followed and will be evaluated stricter than PA1.

Development and Testing

You are **strongly encouraged** to **add tests of your own** as the supplied tests are a subset of the MVP tests. Also remember there are the hidden tests in the ***Field Trial Tests*** that you will be graded on. Try to think about test cases not covered in the supplied tests that might be included in either the MVP or the ***Field Trial Tests*** and add them to the test harness.

The starter files are contained in the `cs30sp22` public folder (`../public` relative to your home directory) in a tarball (a common way to distribute files in Linux). From your home directory, run the following to create the directory PA2 in your home directory:

```
    tar -xvf ../public/PA2.tar.gz
```

Inside the directory at the top-level you will find the following files (and a couple of directories):

Makefile: The Makefile you will use to develop your program. Please examine the Makefile to understand the various targets (especially the test target).

extract.c: This is where you will complete the missing parts of the main() function.

subs.c: This is the file where you will complete the two routines split_input(), wr_row() and insert your helper functions (if any).

misc.c: This is the file the supplied function do_opts() and dropmsg() are. **You should not edit this file.** If you do, you run the chance that your submission will not work as we will test with this version.

misc.h: This is the interface definitions for the functions in misc.c. **You should not change this file.**

subs.h: This is the interface definitions for the functions in subs.c. **You should not change this file.**

The Test Harness

This test harness is basically the same as the one in PA1 (it has the same hierarchy of files). The script runtests are a little longer. Feel free to modify it as required.

In the same PA2 directory you will find another directory called **`tests`**. This is the directory where the test harness is located. There is a shell script in this directory called **`runtests`** (running `make test` in the PA2 directory or in the test directory will call this script). The script runs the tests. In industry this script is more complex (and may be written in another scripting language like python) and often include executable code written to test your program. In the *directory **in*** are **some** of the *Minimum Viable Product* (public tests) for this programming assignment. There may be some corner cases that are not tested in the supplied tests.

In the directory **good** are the expected standard output and standard error output for each test.

In the directory **`out`** are the standard output and standard error output from your program (put there by the script runtests). The stderr output is there for your reference in debugging your program. **We will not grade the output of stderr on this assignment.**

The script **`runtests`** will run your program on each test, capture the standard and standard error output and use the Linux program **`diff`** to compare the expected output (from the good directory) against your programs output (in the out directory), printing the differences (if any). **Please be aware the diff on stderr has been commented out, as we are not grading stderr output. You can check your stderr output if you like by uncommenting the diff in the runtests script.** To read the output of `diff` (also see the man page – `man diff` on the pi cluster):

The script runs

```
    diff expected_output your_programs_output
```

Lines from diff that start with a < is the expected output, lines that start with > is your output

The numbers are the line numbers in the files being compared. The diffs are run both on the standard out and standard error.