CSE30: Computer Organization and Systems SP21 Assignment 3

Important Note:
Please start early so that if you run into any issues, you can seek help! Start early, start often.

Programming in C for Building a Simple ETL program

Captain America has intercepted a secret Hydra database and wants to select specific columns which he thinks is the most vital information from the database. The database is too large for him to do this manually. The Captain seeks your help to write a program that makes this job easier for him to complete.

A common format for moving data between systems is called a **DSV** file. A **delimiter-separated values** (**DSV**) file uses a delimiter (`',' OR '#' OR ' ' OR '\t' ,etc`) to separate values on a line of data. Each line of the file, or "data record", is terminated by a newline (`'\n'`). Each record consists of one or more data fields (columns) separated by the delimiter. Fields are numbered from left to right starting with column 1. A DSV file stores tabular data (numbers and text) in plain text (ASCII strings). In a proper DSV file, each line will always have the same number of fields (columns).

```
MSIDN,IMSI,IMEI,PLAN,CALL_TYPE,CORRESP_TYPE,CORRESP_ISDN,DURATION,TIME,DATE
068373748102,208100167682477,351905149071,PLAN1,MOC,CUST1,0612287077,247,12:07:12,01/01/2012
068373748102,208100167682477,351905149071,PLAN1,MTC,CUST2,0600000001,300,12:15:09,01/01/2012
068373748102,208100167682477,351905149071,PLAN1,SMS-MO,CUST1,0613637193,0,12:18:18,01/01/2012
068373748102,208100167682477,351905149071,PLAN1,SMS-MT,CUST1,0612899062,0,12:21:07,01/01/2012
065978198280,208100310191699,356008289837,PLAN3,MOC,CUST1,0612283725,90,12:00:00,01/01/2012
065978198280,208100310191699,356008289837,PLAN3,MOC,CUST1,0613069656,82,12:02:27,01/01/2012
065978198280,208100310191699,356008289837,PLAN3,MOC,CUST1,0613481951,78,12:04:41,01/01/2012
065978198280,208100310191699,356008289837,PLAN3,MTC,CUST2,0600000001,92,12:07:13,01/01/2012
065978198280,208100310191699,356008289837,PLAN3,MTC,CUST2,0600000002,94,12:09:40,01/01/2012
065978198280,208100310191699,356008289837,PLAN3,MTC,CUST1,0612063352,114,12:12:40,01/01/2012
065978198280,208100310191699,356008289837,PLAN3,SMS-MO,CUST1,0613103364,0,12:13:42,01/01/2012
065978198280,208100310191699,356008289837,PLAN3,SMS-MO,CUST1,0613751973,0,12:14:44,01/01/2012
065978198280,208100310191699,356008289837,PLAN3,SMS-MO,CUST1,0613672843,0,12:15:44,01/01/2012
065978198280,208100310191699,356008289837,PLAN3,SMS-MT,CUST1,0612769488,0,12:16:42,01/01/2012
065978198280,208100310191699,356008289837,PLAN3,SMS-MT,CUST1,0613164676,0,12:17:39,01/01/2012
065978198280,208100310191699,356008289837,PLAN3,SMS-MT,CUST1,0613399901,0,12:18:39,01/01/2012
067599860569,208120276653317,353297808290,PLAN2,MOC,CUST1,0612089847,116,12:00:00,01/01/2012
```

In the example above we have a sample of a Call Detail Record (CDR) in DSV format with the delimiter as a comma. A CDR file describes a cell phone voice call from one phone to another phone. Each record has 10 fields or columns. The first record of the file is a label for that column (field). Each column can be empty, and the last column is never followed by a `','`. It always ends with a `'\n'` for every line record.

An empty Call Detail Record in the above format would have nine (9) commas in it as shown below.

,,,,,,,,,

For the purposes of this assignment, assume the following of the input data:

(1) The DSV file is ASCII text based and can be edited by text editors.
(2) Every line (including the last line) ends with a `'\n'`.
(3) All records have the same number of data fields.
(4) A data field can be empty only when there is more than 1 field in each record.
(5) The default (if the option -d is not provided as input) delimiter is a comma (`','`)
(6) The delimiter can be any ASCII printable character except the `'\n'` (newline) ,`'\'` (backslash) and `'\0'` (NULL) characters
(7) No empty record will be provided. No extra options will be provided as input.

For example, a 3 field DSV has the following variations

- `1#2#3` (delimiter is `'#'`)
- `I am a string_another string_5` (delimiter is `'_'`)
- `@@4` (delimiter is `'@'`)
- `,,` (delimiter is `','`)

Spaces (or lack of spaces) in a field are to be preserved in this assignment.

In this assignment, you will write a program that reads DSV data from standard input, and writes a modified DSV file to standard output. In the description, record columns (fields) are numbered from 1 being the leftmost column, to N being the last column. (N is also the number of columns in a single record.)

Requirements:

1. This program requires a **mandatory** command line flag, `-c` that must be followed by an unsigned integer option argument that shall be one (1) or larger. This is the number of columns (fields) that every valid record in the input data (read from stdin) must have.

2. After the input column specification, there is an **optional** flag `-d` which will be followed by the delimiter for the file within double quotes (eg: `-d "#"`).

3. After that, there must be one or more additional column numbers (positive integer 1 or greater). Each of these arguments specify one of the input record columns, so they must range in value from 1 to the number of columns (fields) in the input file.

4. The order of the fields in the argument list specifies the order the input records are in the output DSV record that is written to standard output (stdout) preceded and followed by the string `"~~Hydra Secret~~"`. [Refer to the sample examples below]

5. All usage and error messages are written to `stderr`.

6. If the mandatory `-c` flag is missing, the usage is to be printed to stderr and the program exits with `EXIT_FAILURE` as a return value from the program. If the `-d` flag is missing, the delimiter is the comma character (`','`)

7. Successful processing will return `EXIT_SUCCESS` from the program.

8. You must use `getopt()` to parse the command line arguments.

Usage:

```
./cnvtr -c input_column_count [-d " "] col# [col#...]
```

Sample Examples :

Example #1

Given an input file with 4 columns containing the following 3 records of data:

```
10,20,30,40
a,b,c,d
this is input,more input,3,last input
```

Calling the program as:

```
./cnvtr -c 4 4 3 2 1 < input_file > output_file
```

It says to read an input DSV file where each record has 4 columns and the delimiter is a comma. The output specification is to write a DSV file where each output record has a column order of 4,3,2,1 from each input record with the delimiter as a comma.

The columns above are in order 1,2,3,4. For e.g.: in the case of 10,20,30,40 – 10 (column 1) is entry 0 in the array.

The output will contain three records that look like:

```
~~Hydra Secret~~
40,30,20,10
d,c,b,a
last input,3,more input,this is input
~~Hydra Secret~~
```

You do not need to parse "<" and ">" from the command line in your program. These are redirection operators and are parsed by the shell before the arguments even arrive in your program.

Example #2

Calling the program, cnvtr with the same input but as:

```
./cnvtr -c 4 -d “,” 3 < input_file > output_file
```

It says to read a DSV file with 4 columns and only write column 3 of the input file to the output DSV file.

The output will contain three records that look like:

```
~~Hydra Secret~~
30
c
3
~~Hydra Secret~~
```

Example #3

Some DSV files want to allow the fields to contain the delimiter. To incorporate this functionality, those are escaped with a '\' (backslash). (Refer to Part 2 of the assignment for more details on this case.)

Given an input DSV file of four records could look like:

```
1#2#test\#string#4
```

When used as

```
./cnvtr -c 4 -d “#” 3 4
```

Output is

```
~~Hydra Secret~~
test\#string,4
~~Hydra Secret~~
```

Description:

1. [40 points] Write a C program that reads DSV data from standard input and writes a DSV file to standard output.

2. [10 points] Handling of DSV files that include the delimiter as part of the values.

Part 1:

Step 0: Getting started

We've provided you with the starter code at the following link:
https://github.com/cse30-sp21/A3_starter

1. Download to get the `Makefile` and `README`.
2. Fill out the fields in the `README` before turning in.
3. Open your favorite text editor or IDE of choice and create new files named `converter.c` and `converter.h` .

   * **The files you turn in must be named as specified or else the autograder and Makefile provided will not work.**

The Makefile provided will create a `cnvtr` executable from the `converter.c` file you will write. Run it by typing `make` into the terminal. By default, it will run with warnings turned on. Run `make no_warnings` to run without warnings. Run `make clean` to remove files generated by `make`.

Step 1: Setup

Setup 1: Parse the command line options make sure you get the `-c` col and the `-d` "delim" (if provided.)

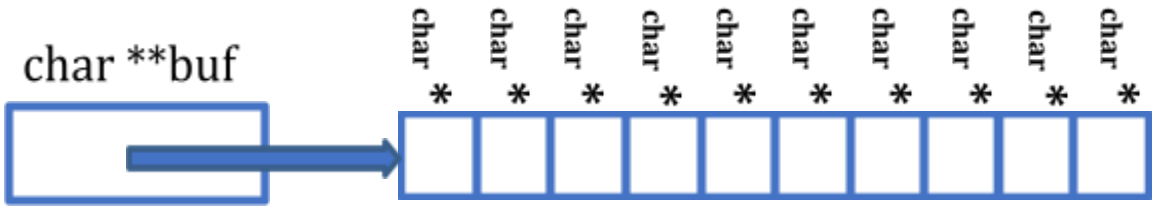
This col will be the number of fields in the input file. The delim will be the delimiter for that input file. On the command line there must be one or more additional args. These args specify field numbers in the input file. You will use this value to validate that the correct number of fields (columns) are in each line input record. You will need to verify all the specified args for output range in value from 1 to the number of fields in each input record (the value associated with the -c flag). If there is an error, print the error message to stderr alone with usage and exit (EXIT_FAILURE).

Setup 2: Build two arrays, one for input processing and one for output processing.

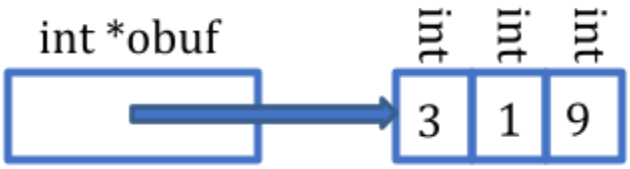
For input, you will need to read a line of input using `getline()` (see `man 3 getline`) and break it into tokens. Each token is delimited by a delimiter or a newline `'\n'` in the input record buffer. Using **`malloc()`**, allocate an array of pointers to `char`s (**buf in the picture below). The number of array entries is the same as the number of fields in the input file. Each entry will point

to the start of each field. This array is similar to `*argv[]`, except we do not need to null terminate it with an empty entry.

As an example, say we are processing records that have 10 input fields (like the CDR record above), the input processing array will look like this.



For output, you will need to create an array of **`int`** using `malloc()`. The number of elements in the array is equal to the number of fields that will be written to the output. This array's size is determined by the number of args passed on the command line after the **`-c or -d`** flag. Each entry will contain the column number index to be output, and the location of the column numbers to be printed in the array from index 0, is the order the columns to be written. Let's assume there were three output column arguments: 3,1,9.



Here is how you can allocate an array of int pointers and an array of char:

```c
int main(int argc, char *argv[]) {

    // to allocate an array of 20 integer pointers (int* )
    int **pptr;
    pptr = malloc(20 * sizeof(int *));
    // pptr points to an array of 20 entries,
    // each entry can store a pointer
    // to an int
    //when we are done with this array, we free it
    free(pptr);

    //to allocate an array of 15 characters (char)
    char *cptr;
    cptr = malloc(15 * sizeof(char));
    //cptr points to an array of 15 entries,
    //each entry can store one character
    //when done with the array, we free it
    free(cptr);
}
```

Step 2: Processing

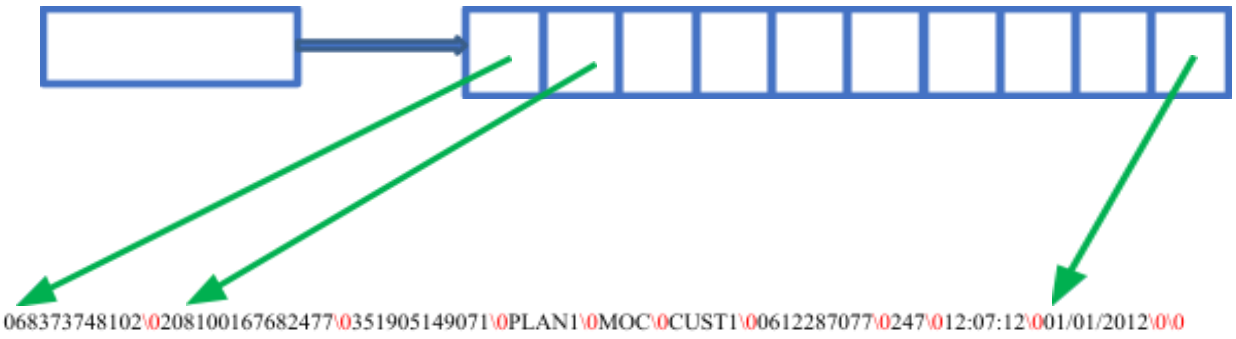
Processing 1 :

Read one line from **`stdin`** using `getline()`. Set the pointer in the input array (char** buf) to the start of the input buffer. Walk the input buffer using pointers only, looking for the delimiter or a `'\n'`, and making sure to stop at `'\0'`. For each `delimiter` 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 (char** buf) entry. 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 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 an array filled with pointers to the start of each field (column) in the record and each field is a properly `'\0'` terminated string.

Using the CDR example again, say that we start with this DSV input buffer after it was filled by getline(). The red highlight of newline and the null are really just two chars (shown with the \escape as an illustration)

068373748102,208100167682477,351905149071,PLAN1,MOC,CUST1,0612287077,247,12:07:12,01/01/2012\n\0

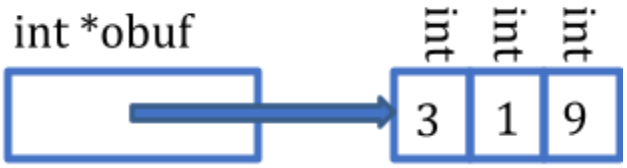
After processing the DSV record buffer (the one filled by `getline()`), we have the following data structure alignment between the input array and the DSV buffer.



You can see each delimiter (`','`) in the DSV record buffer, as well as the `'\n'` at the end, has been replaced by a NULL `'\0'`. Each entry in the input array of pointer (char** buf) to chars points at the start of a field in the record. `buf[0]` points at field 1, etc. All the entries in the array are updated each time a new record is processed.

Processing 2 :

Once you have your array of pointers to fields filled for one record, you will need to write the output. Walk down the output array from index[0]. Each entry in the output array contains the index number of the next field to write to standard output. Use this number as an index into the first array to get the pointer to the input field you need to output (watch your indexes, they start at 0 and columns are numbered from 1). Remember your output must also be a correct DSV format.



Using the output buffer as above, we would write the fields to the output by doing a `printf` directly from the input buf of pointers. We have 3 records in our output that select buf[2], buf[0], and buf[8] pointers from the input array, after mapping fields numbers to array indexes (subtracting 1). Notice we do not have to copy any strings, use `strlen()` or allocate more space beyond the two arrays at program start.

Processing 3:

Loop to **Processing 1** and repeat for the next record. When the end-of-file character EOF is reached, return properly with `EXIT_SUCCESS`.

Understand `getline()` and implement a similar loop in the code to a file titled `converter.c`.

```c
char *buf = NULL;       /* input buffer pointer allocated by getline(). see man3 getine() */
size_t bufcap = 0;      /* size of buffer pointed at by buf. see man3 getine() */

/*
 * read the input line at a time, break into tokens and write out the selected columns
 * getline will allocate and adjust buf size as needed and will always properly terminate
 * teh input with a '\0'
 * getline leaves the trailing '\n' in the input buffer if it is there
 *
 * when executed interactively from a terminal,
 * to signal EOF (where getline returns a -1) type cntrl-d on a line by itself
 * this is not needed on input froma a file or when input is piped from another process
 */

linecnt = 0;

while (getline(&buf, &bufcap, stdin) > 0){
        linecnt++;
        // Try below 2 steps inside this loop
        // STEP-1 : Process the input and check if it has the proper number of fields
        // STEP-2 : Write the selected fields to the input

}
```

Note: This snippet of code **doesn't** give any idea of actual implementation details. This is just a reference of how getline() works. For more details, Please visit `man 3 getline`.

To compile and run your code:

1) put the code in a file (e.g. `converter.c`).
2) run `make` (alternatively you can run `gcc converter.c`) to create an executable.
3) call the executable with `./[name of executable]`.

For each function, other than main, you should have the function prototype in a file titled `converter.h`. Be sure to have `#include "converter.h"` at the top of your `converter.c` file.

Once you can do that, you are ready to work on the assignment. Modify your program to create a program that reads CSV data from standard input and writes a CSV file to standard output.

You will receive 40 points for the primary converter program.

Part 2:

Some DSV files want to allow the fields to contain the delimiters. In such cases, the delimiter is escaped with a '\' (backslash) [10 points]

Given an input DSV file of four records could look like
`1,2,test\,string,4`

When used as
`./cnvtr -c 4 3 4`

Output is
```
~~Hydra Secret~~
test\,string,4
~~Hydra Secret~~
```

Note: You should only choose from the following library functions: `malloc()`, `printf()`, `fprintf()`, `getopt()`, `free()`, `atoi()`, `getline()` and `exit()` in your code.
Note: function prototypes should be written in `converter.h`

Style and Commenting

No points are explicitly given for style but teaching staff won't be able to provide assistance or regrades unless code is readable. Please take a look at the following Style Guidelines