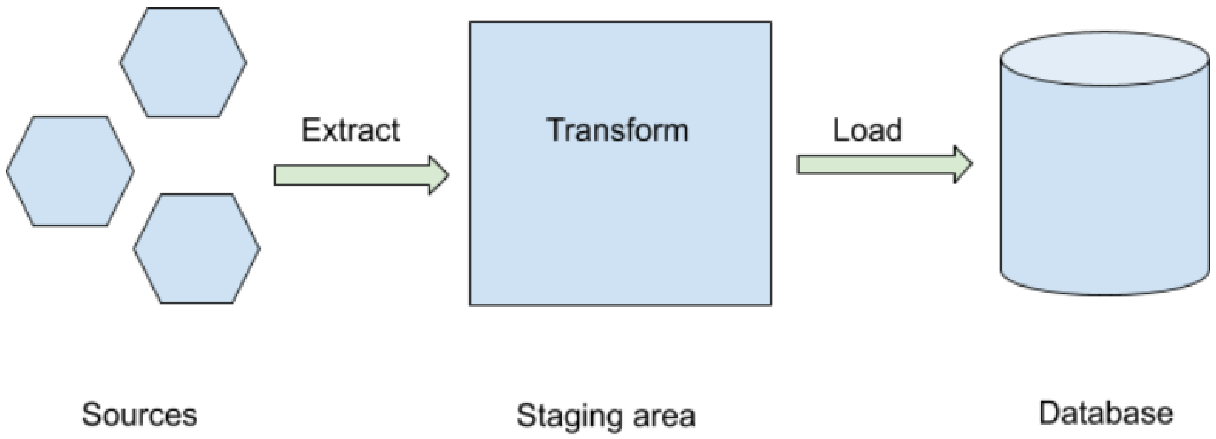
CSE30: Computer Organization and Systems FA20
Assignment 3
Instructors: Leo Porter, Keith Muller

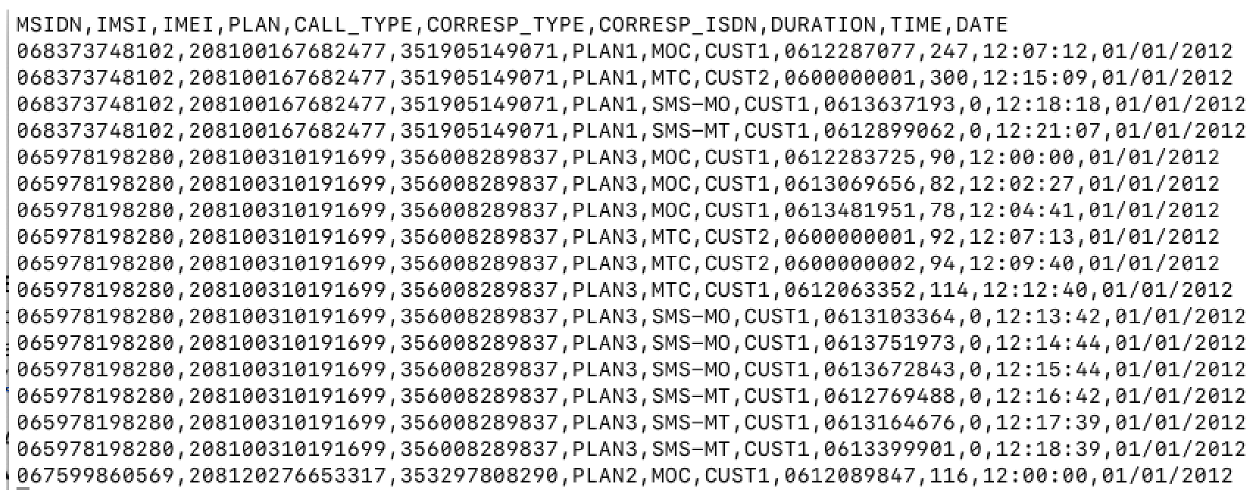
Intro:
Programming in C for Building a Simple ETL program

The movement of data from a source system to a destination system often requires the data to be reformatted to meet the input requirements of the destination systems. Source systems range from sensor data from a hardware system (like solar panel operating data) to rows extracted from a database system. Target systems can be other devices or a different database system. In the world of enterprise data management this data movement process is broadly called ETL, Extract, Transform and Load processing. ETL software may be purchased from vendors but is often custom written to meet the unique requirements of the output data format of the source systems and the input data format of the target system.



Extract processing (reads) data from standard input system (directly from the source system or a “spool” file created by the source system). **Transform processing** works on the input data and may include operations to validate, modify and reordering the data input. **Load processing** (writes) the transformed data to the target system via standard out.

A common format for moving data between systems is called a **csv** file. A **comma-separated values** (**CSV**) file uses a comma (‘,’) to separate values on a line of data. Each line of the file is terminated by a newline (‘\n’) called a data record. Each record consists of one or more data fields (columns) separated by adjacent commas. Fields are numbered from left to right starting with column 1. The use of the comma as a field separator is the source of the name for this data format. A CSV file stores tabular data (numbers and text) in plain text (ascii strings) and in a proper CSV 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 in CSV format. 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 CDR record would have nine (9) commas in it as shown below.

```
,,,,,,,,,
```

For the purpose of this assignment you can assume the following on the input data:

(1) That the CSV file is ascii text based and can be edited by any text editor
(2) that every line ends with a '\n'
(3) A data field can be empty only when there are more than one field in each record.

For example, a three field CSV has the following variations

```
1,2,3
I am a sting, another string,5
,,4
,,
```

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

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

Requirements:

1. This ETL program requires a single 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. Records that do not have this column count will be dropped and not be written to the output.

2. After the input column specification, there must be one or more additional column numbers (unsigned 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 in the input file.
3. The order of the fields in the argument list specifies the order the input records are in the output csv record that is written to standard output (stdout).
4. All usage and error messages are written to stderr.
5. When a record does not contain the correct number of fields, the record is dropped, and a message is written to stderr as such:

   ./cnvtr: dropping record# *number*

   where *number* is the line (record) number of the input where the first input record (line) is starts with the number 1

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.
7. Successful processing (ignoring dropped records) will return EXIT_SUCESS from the program.
8. You must use getopt() to parse the command line args.

Usage is

```
      %./cnvtr -c input_column_count col# [col#...]
```

Example #1

Given an input file with four (4) columns and three records containing the following three 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
```

Says to read an input CSV file where each valid record has 4 columns. The output specification is to write a CSV file where each output record has a column order of 4,3,2,1 from each input record.

The columns above are 1,2,3,4. For eg: in the case of 10,20,30,40 -- 10 is entry 0 in the array.

<u>The output contains three records:</u>

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

Example #2
Calling the program, cnvtr with the same input but as
%./cnvtr -c 4 3 <input > output
Says read a CSV file with 4 columns and only write column 3 of the input file to the output

<u>The output contains three records that looks like</u>
30
c
3

Some CSV files want to allow the fields to contain commas. Each field can optionally be enclosed in quotes.
Given an input CSV file of four records could look like:
1,2,"test,string",4

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

<u>Output is</u>
"test,string",4

ETL programs are measured on runtime performance and you should keep this in mind when writing your code.

Description:

1. [8 points] Write a C program to create a program that reads a CSV data from standard input and writes a CSV file to standard output.
2. [2 points] Handling of CSV files that include commas.

Step 0: Getting started

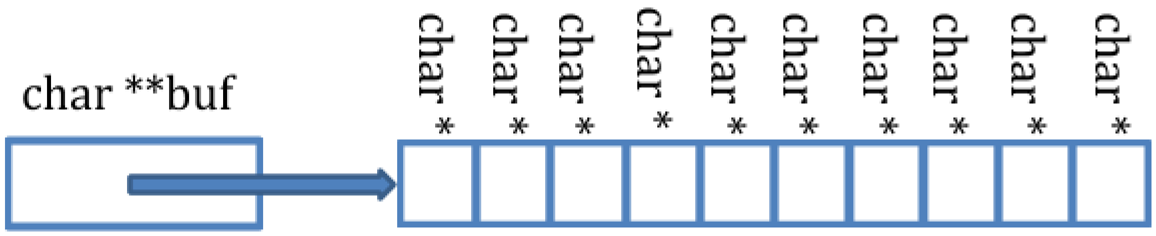
We've provided you with a Makefile here. Download the Makefile to use.

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.**

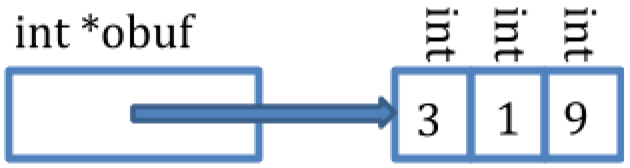
The Makefile provided will create a `cnvtr` executable from the `converter.c` file you will write. Run it by typing `make` into the terminal. By default, it will run with warnings turned on. Run `make no_warnings` to run without warnings. Run `make clean` to remove files generated by make.

Step 1: Setup
Setup 1: Parse the command line options make sure you get the -c col.
This col will be the number of fields in the 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 of 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 either a ‘,’ or a newline ‘\n’ in the input record buffer. Using malloc() allocate an array of pointers to chars. 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 the same as *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 ints 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 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. Lets assume there were three output column arguments 3,1,9



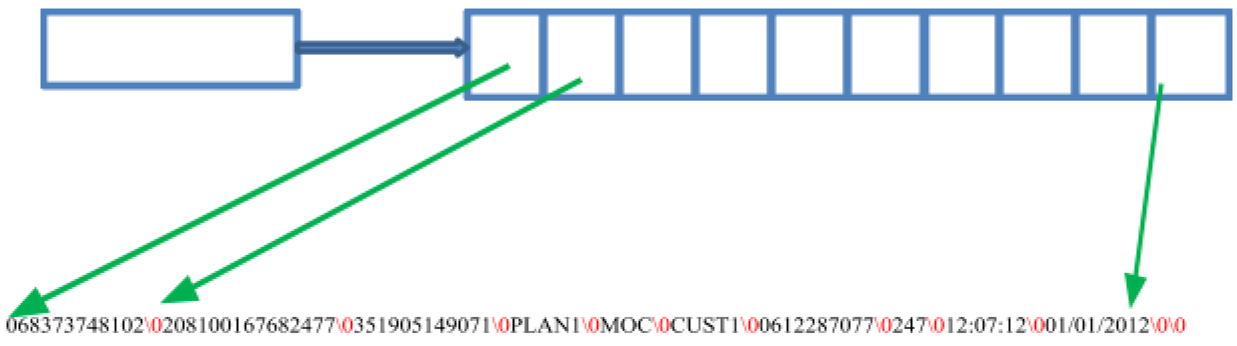
Step 2: Processing
<u>**Processing 1**</u>
Read one line from stdin using getline(). Set the pointer in the input array to the start of the input buffer. Walk the input buffer using pointers only, looking for a ‘,’, a ‘\n’, making sure to stop at ‘\0’. For each ‘,’ or the final NL ‘\n’ after the last record, replace the , or NL with a ‘\0’ to

terminate that token. Store the pointer to the next char in the next input array entry. Repeat this process until you either reach the end of the input line or fill all array entries. Make sure you throw out any input record (and print a message with the record # to stderr) if there are too few or too many fields in a record. You should not need to zero out the array on each pass, you just keep track of which element pointer is being processed. Do not waste cpu cycles zeroing out arrays when it is not needed. 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 we start with this CSV input buffer after it was filled by getline(). The red highlight of newline and the null are really just to 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 CSV record buffer (the one filled by getline()), we have the following data structure alignment between the input array and the CSV buffer.

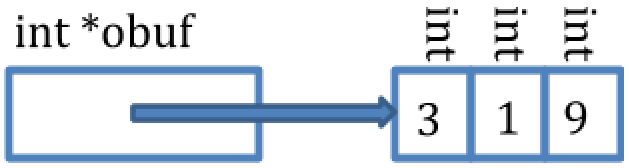

068373748102\0208100167682477\0351905149071\0PLAN1\0MOC\0CUST1\00612287077\0247\012:07:12\001/01/2012\0\0

You can see each ',' in the CSV record buffer has been replaced by a NULL as well as the trailing '\n' at the end has been replaced by a NULL. Each entry in the input array of pointer to chars points at the start of a field in the record. Array[0] points at field 1, etc. If an input record is ok, all the entries in the array are updated each time a new record is processed.
Here is a warm up exercise for the input array here, pointing into the getline buffer with each field null terminated. Add the following code to a file titled `converter.c`

```
        char *buf = NULL;               /* input buffer pointer allocated by getline() see man 3 getline() */
        size_t bufcap = 0;              /* size of buffer pointed at by buf. see man 3 getline() */
```

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 CSV format.

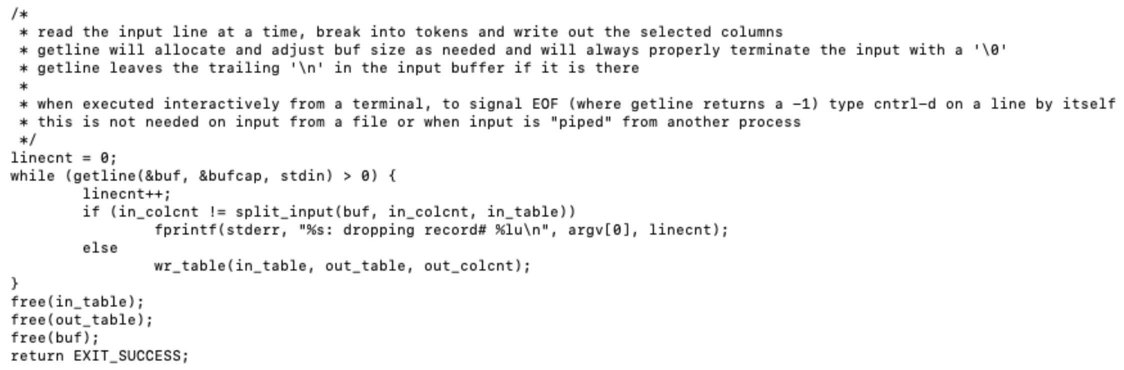


Using the output buffer as above we would write the fields to the output by doing a printf directly from the input buf of pointers. We have three records in our output that select buf[2],buf[0],buf[8] pointers from the input array after mapping fields numbers to array indexes (subtracting 1). Notice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

Processing 3

Loop to processing 1 and repeat for the next record. When EOF reaches terminate properly with EXIT_SUCCESS even if there are dropped lines.

Here is a warm up exercise for output array referencing the input array. Add the following code to a file titled `converter.c`



```
/*
 * 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 the 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 from a file or when input is "piped" from another process
 */
linecnt = 0;
while (getline(&buf, &bufcap, stdin) > 0) {
        linecnt++;
        if (in_colcnt != split_input(buf, in_colcnt, in_table))
                fprintf(stderr, "%s: dropping record# %lu\n", argv[0], linecnt);
        else
                wr_table(in_table, out_table, out_colcnt);
}
free(in_table);
free(out_table);
free(buf);
return EXIT_SUCCESS;
```

To compile and run this snippet of 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 complete the assignment. Now modify your program to create a program that reads a CSV data from standard input and writes a CSV file to standard output.

You will receive 7 points for the primary converter program.

Part 2: Some CSV files want to allow the fields to contain commas. Each field can optionally be enclosed in quotes. [2 points]

Given an input CSV file of four records could look like
1,2,”test,string”,4
When used as
%./cnvtr -c 4 3 4
Output is
“test,string”,4

Note: You should only choose from the following library functions: printf(), fprintf(), getopt() 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

Submission and Grading

1. Submit your files to Gradescope under the assignment titled Homework3. You will submit the following files:

   ```
   converter.c
   converter.h
   README
   ```

   To upload multiple files to gradescope, place all files in a folder and upload the folder to the assignment. Gradescope will upload all files in the folder.

2. After submitting, the autograder will run a few tests:
   a. Checks that all required files were submitted
   b. Checks that `converter.c` compiles
   c. Runs some tests on `converter.c`

Make sure to check the autograder output after submitting! We will be running additional tests after the deadline passes to determine your final grade.

The assignment will be graded out of 10 points, with 8 points allocated to the main Converter program (part 1), and 2 points allocated to csv with commas (part 2). Make sure your assignment compiles correctly through the provided Makefile on the ieng6 machines. **Any assignment that does not compile will receive 0 credit for part 1&2.**