Programming Assignment 3 - part II
Adding Features to a Table-Processing Code Base
Due: ~~MONDAY~~ Tuesday March 15 @ 11:59pm (~~tentative~~).

You have been provided an implementation of a program for sorting tables of data. The given code is a sample solution to a CS211 programming assignment from a previous semester.

The handout for the original CS211 assignment is an Appendix to this document. Before continuing, go to the Appendix and read the assignment -- think about how you might have approached it along the way.

Ok, you're back? The given code base is organized into three files:

tbl.h	Specifies types and function prototypes for creation and manipulation of table data structures. By convention, these functions are prefixed with **tbl_**
tbl.c	Gives implementation of functions specified in tbl.h Together, these behave like an ADT (although the types are not "hidden" as in previous examples, client programs should operate through the specified functions to the degree possible rather than directly accessing the data structure unless necessary -- and then just for "read" operations).
tsort.c	A client program which behaves as specified by the CS211 handout.

	Almost all of the work is done by the tbl_ functions. Most of the code in tsort.c does command line parsing and some simple error checking.
minima.c	See discussion
rowcmp.c	See discussion

Your Tasks

TASK 1: Column Labels and Row Names

Currently, there is no provision for either column labels (text) or row names (also as text).

This seems pretty limiting. So the input format has been modified as follows.

ROW NAMES:
Each row of the table will have a "name" associated with it. This name is simply a string and will always be stored in column 1.

The input remains line-based, but now each such row begins with a string (the name of that row) followed by a colon (":") after which appears the remaining columns (all of which are numerical as before).

For example, the last row from an input in the appendix could be modified as below.

```
Ford Mustang: 1967    20000   10000
```

The row name is by default, column 1 (and, remember, is the only non-numeric column).

COLUMN LABELS:
The first non-empty, non-comment line of the input now

specifies the column labels in sequence, separated by colons (":").

This line also determines the number of columns for the table (almost in the same way the first non-empty, non-comment line does presently).

The first column (reserved for row "names" -- see above) must also have a label.

EXAMPLE:
Completing the transformation of a sample table file from the Appendix, we might have this:

```
# car database
Make and model:   year:   price:  miles
Hyundai Excel:    1989    500     222000
 Pontiac Aztek  : 1997    1500    180000
Pontiac Firebird: 1976    4000    20000
Ford Mustang:     1967    20000   10000
```

Some comments, details and suggestions:

- **Parsing utilities:** The strtok library function is very well suited to the task of extracting "tokens" from lines like the column label line. (The given implementation does not use strtok -- sscanf was sufficient for extracting white-space separated floating point numbers). The source directory for this assignment includes a toy program called strtok.c illustrating a basic usage of strtok.
- **Trimming leading and trailing whitespace for names/column labels:** The input file is likely to be formatted for easy viewing -- things aligning in columns. As a result, row names and column labels may be "padded" with leading and/or trailing white space. Consider the column labels in the example above. The 2nd row name should be "Pontiac Aztek" right? You want to represent it internally as such -- not as

" Pontiac Aztek " (consider the effect leading spaces would have on sorting by row name…). The source file tbl.c contains a template for a recommended utility function to trim leading and trailing whitespace from a string (implementation is up to you…).

- **Why colons?** You might wonder why the numerical values are "white-space separated", but colons are used to separate string tokens from other tokens. Good question. The rationale is that by using a colon, strings can include spaces (e.g., "Ford Mustang").
- **Label and Name Length Limit:** Let's put an upper limit on allowable length of these strings (row names and column labels): 30 seems like plenty! If an input file contains a row name or column label longer than 30 characters, consider this a format error and terminate reading the input as with other format errors. To simplify matters even further, this rule can be applied to column labels / row names ***prior to*** any stripping of leading or trailing whitespace (see above).
- **Row names do not need to be unique:** if the user has multiple rows with the same name, so be it -- this is not an error.
- **Sorting by name allowed:** Remember that row names are always column-1. This impacts the sorting operation -- a user of the tsort program can sort the table according to row names if they so choose. You will have to figure out how to handle this via the qsort library function.
- **Where to store row name?:** the Row structure currently contains an array of type double to store a row's numerical elements; how can you put a string in index-0 of this array? Well, you *could* do this by changing the element type to an appropriate C union. But I recommend that you simply store the row name as a separate field in the Row structure.
- **Column labels and row names required - even if empty:** A user may not want to give meaningful row names

and/or column labels. This is fine, but they still need to specify them as empty strings. For example:

```
# empty column headers
# and empty row names
:::
:1967    20000   10000
:1976    4000    20000
:1989    500     222000
:1997    1500    180000
```

TASK 2: Eliminating Row Upper-Bound

The given implementation imposes an upper bound of 100 on the number of rows in a table. Seems rather arbitrary and impractical no?

You will modify the implementation to have no fixed bound. You already know a technique for this… right? (We will leave the maximum number of columns at 20…)

TASK 3: Lexicographic Sorting

The tsort program sorts the given table by a specified column (given by the -c flag). You will extend the implementation so that multiple columns can be specified where the first specified column determines the "primary" sorting criteria; if multiple rows are identical in the primary column, the ties are broken by the 2nd specified column and so on.

For example,

```
$ ./tsort -c 2 4 < input.txt
```

Will produce the table ordered by column-2 where blocks of rows identical in column-2 are ordered by column-4.

Hint: since the sorting strategy already implemented guarantees "stability", lexicographic sorting can be accomplished without too much trouble. Even though qsort (which is not, in general, stable)is used, the implementation ensures the result is stable -- did you notice how?

The user can specify any sequence of columns they like after the -c flag (including column-1 -- the name column). You need to work out the logic of parsing the command line arguments.

TASK 4: Determining the "Minima" of a table

A table is a collection of "tuples". Let's suppose that each numerical tuple entry is a trait that we prefer to be small. For example, suppose each row in a table captures an option for taking a trip: one column gives the cost of the option and another gives the travel time. Presumably you'd like to spend less money and get to your destination as soon as possible. Suppose your travel agent offers you these options:

A. (cost: $200)(time: 3 hours)
B. (cost: $150)(time: 2.5 hours)
C. (cost: $175)(time: 2 hours)
D. (cost: $400)(time: 0.5 hours)
E. (cost: $400)(time: 0.75 hours)
F. (cost: $150)(time: 2.5 hours)

Presumably, if you are willing to spend more money, you should get a benefit -- in this case less travel time.

When comparing two options X and Y, there are 4 possibilities:

- X and Y are **identical** in all dimensions of interest.
- **X *dominates* Y**: X and Y are not identical and X is as good or better than Y in every dimension.
- **Y *dominates* X**: X and Y are not identical and Y is as good or better than X in every dimension.
- Otherwise, **X and Y are incomparable** -- X is better in some dimensions and Y is better in others

With respect to the 4 travel options above:

B dominates A
D dominates E
F and B are identical

Options A and E are inherently sub-optimal. One of F or B can be considered redundant.

So, let's eliminate A, E and F (or B) from our options. This leaves us with options B, C and D.

Observe that:

B and C are incomparable
B and D are incomparable
C and D are incomparable

We say that B, C and D are a *minima* of the given table (and also form another table…). They are all non-dominated and there are no duplicates.

Intuitively, a minima captures a tradeoff between different alternatives (sometimes called a pareto curve).

Formally, the minima of a table (collection of tuples) is the set of all non-dominated entries.

Comments: contrast this criteria with that of the "lexicographic sorting" above. Here, no column is considered more important than any other, while lexicographic ordering implies an order of importance.

Also, notice that the notion of dominance applies no matter how many columns are in a given table.

Finally, Your jobs:

- Implement the cmp_rows function as specified in tbl.h.
  - Comparison is performed on all columns in the table except the row name.
  - Take note of the enumerated type defined for the possible outcomes.
- Implement the minima function as specified in tbl.h
  - This function has no runtime requirements.
  - It may operate by calling cmp_rows (multiple times) to ultimately determine a minima.
  - The non-dominated rows are used to populate a new table which is returned. Rows are completely copied (no pointer alias).
  - The new table **retains the column labels** as the original table (i.e., when printed).
- (Not a TODO) - you have been given a program minima.c which reads a table from standard input, calls the

tbl_minima function and prints the resulting table to standard output.

- (Not a TODO) - you have also been given a program rowcmp.c which reads a table from stdin, performs all pair-wise row comparisons and displays the results in a grid.

Submission Instructions - ~~forthcoming~~.

Submit through blackboard a zipped tar file of your source directory.

There should be a Makefile supporting the following targets (some of which you already have been given):

- minima
- rowcmp
- tsort
- tbl.o

In addition, you may include a readme file if there is anything you think the grader should know.

APPENDIX: Handout For Original tsort assignment (CS211-summer-2015)

Overview

Suppose you are given as input a table of numerical data. Each row of data is a "record" or "tuple". For example, we might have information about used cars in the table. The first column could be year; the second price; the third mileage.

The table below has data for four cars.

```
1989    500     222000
1997    1500    180000
1976    4000    20000
1967    20000   10000
```

Given such a table we'd like to be able to sort it by any of the three columns.

Sorting by year would give:

```
1967    20000   10000
1976    4000    20000
1989    500     222000
1997    1500    180000
```

Sorting by price would give:

```
1989    500     222000
1997    1500    180000
1976    4000    20000
1967    20000   10000
```

Sorting by mileage would give:

```
1967    20000   10000
1976    4000    20000
```

```
1997    1500    180000
1989    500     222000
```

Your program will perform this kind of sorting.

Basic functionality:

Your program will read table data from stdin. The usage is as follows (from the shell):

```
% tsort -c <column-num> [-d]
```

where <column-num> is a placeholder for an integer column specifier telling the program which column to sort by. **Columns are numbered starting at one, not zero!** Rationale: we programmers like to start things at zero, but the rest of the world likes things to start at one. Our users are not necessarily programmers themselves.

> Note: the program needs to read command-line arguments (i.e., using the argc and argv parameters to main); if you have never done this before, don't worry -- it's not difficult; here is one of many online resources:
>
> http://en.wikibooks.org/wiki/A_Little_C_Primer/C_Command_Line_Arguments

The "-d" flag is optional (the square brackets are used to indicate that it is optional) and indicates that the sort should be in ***decreasing order*** (or technically non-increasing order). In default mode -- i.e., the -d flag is not specified -- the sort is performed in increasing order.

As stated, the program reads the table from stdin but this doesn't mean that the user has to type in the table each time; the UNIX shell has "redirection" operators which let you connect stdin to a file. For example,

```
% ./tsort -c 5 < inp.txt
```

would connect tsort's stdin to a stream of characters from the file inp.txt. The tsort program itself is completely unaware of this redirection -- it is just reading from its stdin.

Important: the text "< inp.txt" typed in at the shell is not part of the command line arguments to the tsort program. You can verify this by examining the fields in argv when the program starts.

Another example:

```
% ./tsort -c 2 -d < inp.txt
```

Would sort the input according to column two in *decreasing* order.

Input format:

- Every ***row*** of the table is contained on a ***single line of the input.***
- Values on each line are ***white-space separated*** -- no need to fuss with comma separated values.
- Each input line can have at most 200 characters (a full-screen terminal window on my machine with 12 pt. font has 204 columns). This allows straightforward line-based input parsing. A new line character at the end of the line does not count against the 200 char budget.
- The maximum number of columns is 20. This saves you the trouble of dynamic allocation and re-allocation.
- We allow (for now) at most 100 rows.

Input Parsing and Error Handling.

Blank lines: blank lines (i.e., lines that contain only white space) are simply skipped and no error is reported. Blank lines do not correspond to a row in the table.

Comment lines: lines beginning with # (in the first column) are comment lines and are simply skipped.

Below is a list of input errors. When detected, the program prints an error message to stderr and terminates. The error message should include the error type and the line number of the input on which it occurred (line numbers start at 1):

Input errors (cause termination of program after error message):

- Lines longer than 200 characters.
- Lines with non-numeric data: lines with strings that cannot be parsed as floating point numbers.
- Lines with more than 20 numeric values. Such lines may have fewer than 200 characters, but exceed our pre-specified column limit.

Determining number of columns:

- The first non-empty / non-comment row of the input determines the number of columns for the table as a whole -- i.e., the number of columns is ***implicit.***
- If any subsequent row has more or fewer entries, this is an **input error.** As with previous errors, you print a message to stderr and terminate the program. The error message should include the line number of the error.

Command line arguments:

If the program is run with no command line arguments, you will assume that you should sort according to first column.

To sort based on a different column, the user uses the -c flag. For example,

```
% tsort -c 3 < input.txt
```

would be used to sort the table by column 3. ***Recall that column numbering begins at one, not zero!***

Error condition: if the specified column is out of range, an error is reported and the program terminates.

Output:

On a successful run, the program writes the re-ordered table to stdout. The requirements are:

- The output should be "pretty" -- i.e., all columns should

be nicely aligned. If you don't know how to do this, study up on some of the printf specifier options.

- The output must be parseable by the tsort program itself. For example, you should be able to do things like

```
% tsort -c 2 < tbl1.txt > tbl2.txt
% tsort -c 1 < tbl2.txt > tbl3.txt
```

As previously mentioned, we can use "<" to redirect the contents of a file to stdin of a program; similarly, we can use ">" to redirect the stdout of a program to a file.

Requirements, Tips and Challenges:

- **(Requirement) You must use qsort:** From the week 3 lab, you know (or will know) something about function pointers in C and how, for example, they can be used to make the qsort library routine general purpose. **Your tsort program must use the qsort routine to perform the sorting.**
- **(Tip) Think carefully about your data structure layout:** the sort will shuffle around "rows." What is a row in your data structure and how can it be moved? Can you avoid copying all of the elements of the row? (Yes you can!) Once you've figured that out, what is the *size* of the elements from the point of view of qsort?
- **(Tip/Requirement) Encapsulation:** You should think carefully about what a table is and bundle it all up in a single data structure -- e.g., you might have a struct (e.g., tbl_struct) through which you access everything about a table. You should then come up with a list of operations you'd like to be able to perform on tables and write functions corresponding to these operations. You should think about separating this out into a separate file. In this strategy you would have a small driver program which mostly calls routines for operating on tables (and written in another file).
- **(Challenge) Getting the comparator function to work for different columns:** You need to be a little creative here! You don't want to have to write 20 different comparator functions! You need to figure out a way to dynamically configure the column on which to sort and make it so that your comparator function has access to it.

Suggestions/ideas:

- A **global variable** which indicates the column to be sorted. Prior to calling qsort, you would set this variable to the desired value. Your comparator function would then access it to perform the comparison. This is a so-so solution since as you know we like to avoid globals.
- Similar idea, but organize your data structures in such a way that the column to be sorted can be accessed through the "things" being sorted themselves. Here, the "things" are rows -- think about augmenting the row data structure in such a way as to give access to the column by which you are sorting. This value need not be global, but rather "owned" by the table data structure itself.

Bonus

You will receive up to 15% extra credit if your sort is *stable*. When is a sort stable?

> Suppose two elements a and b in an input are equal according to the comparator function and that element a appears before element b in the input sequence. *A stable sort will always place a before b in the output sequence.*
>
> In other words, entries that are tied appear in the same *relative order* as they did in the input.

Why might this be useful? Returning to the initial used car example, suppose I want the cars listed in increasing order of year, but I want all cars from the same year to be ordered by price. If I first sort by price and sort the result by year, what happens?

```
% tsort -c 2 < cars.txt > cars_by_price.txt
% tsort -c 1 < cars_by_price.txt > cars_by_year.txt
```

If the sort is stable, two cars from the same year, but with different prices will appear (in cars_by_year.txt) in the same order as they were in the cars_by_price.txt table (from the first sort).

```
Kind of handy.
```

Tips and Resources

General Recommendation
When trying out a library function you haven't used before (like `qsort, fgets,` or `strtok` discussed below) write a simple, small program or two to test it out! For example, before trying to use `qsort` in your tsort implementation, write a program that uses `qsort` to sort an array of some simple type (`int` for example); then maybe try something a little more complex.
Reading input
The input format is specifically "line-by-line." So we highly recommend that you read and parse each tuple in two phases: 1. Read the next line into a buffer (array of char). 2. Then parse the individual fields in the buffer. **Reading a line into a buffer:** Use the **`fgets`** function for this step! AVOID the **`gets`** function. It is not "safe". Background reading: ● Chapter 2 of "Expert C Programming", "Sins of Mission" section. ● Man page for `fgets`: http://man7.org/linux/man-pages/man3/fgets.3.html Remember that the program spec puts a limit on the length of an input line and that you will have to detect when a line exceeds this limit.

Parsing the fields in a buffer:

You will probably find the **`strtok`** ("string tokenizer") library function useful to identify the individual fields in an input line.
Background reading:

- `strtok` man page: http://man7.org/linux/man-pages/man3/strtok.3.html

The individual tokens will still need to be parsed into numerical values. The **`sscanf`** library function is recommended for this task.
Background reading:

- **`sscanf`** man page: http://man7.org/linux/man-pages/man3/scanf.3.html

Working with `qsort`

Remember that you are required to use the general purpose **`qsort`** library function for this assignment.

You need to have a working understanding of C function pointers and passing them as parameters to accomplish this.

Background Reading:

- Man page for `qsort`: http://man7.org/linux/man-pages/man3/qsort.3.html
- Illuminating discussion involving **`qsort`**:
  Chapter 8 of ***"Expert C Programming" -- "How and why to cast" section.***
- Fundamentals of function pointers:
  Chapter 3 of ***"Understanding and Using C Pointers" -- "Function Pointers"*** section.