NAME:
    MULTITHREAD SORTER

DEVELOPERS:
    Justin Chan
    Forrest Smith

OVERVIEW:
    multiThreadsorter.c will sort csv file(s) (comma separated value file) by a specified column. The entries are sorted via mergesort.

GOALS:
    Sort a single CSV file, or a directory of CSV files, on a given column title.
    Take user-specified input and output directories.
    Sort movie-based, properly formatted CSV file containing the specified column.
    Return number of threads and their ID's, and initial process ID
    Combine all proper CSV's, sort their entries, and output to "AllFiles-sorted<fieldname>.csv"

SOLUTION:
    - The sorted rows are then written to "AllFiles-sorted<fieldname>.csv" to the proper directory.
    - Sorting all rows is accomplished via mergesort (mergesort.c). Our comparator functions are based on NUMBERs and STRINGs (enums for validating what type of data we are sorting on), and they are passed into our mergesort function along with the column of the current CSV file.   - CSV files and output files are read/created with open(), parsed, sorted, and subsequently closed with close().
    - All rows of all valid CSV's are stored in a linked list of _Row objects to be sorted
    - CSVs are parsed into column header structs and row structs. The column header structs hold the columns of the CSV file and the type of data held in the column, using a _format enum. The rows hold the cells of each row of the CSV file in a string (char**) array.
    - The column headers of each file are checked to ensure that they adhere to our targeted set of headers. If an additional un-targeted column is present, or our column to sort by is absent, we do not use the file (thread count still incremented).
    - CSV files and output files are read/created with open(), parsed, sorted, and subsequently closed with close().
    - Each row of the CSV is read in using the read() function.
    - Folders are traversed and tracked with directoryHandler() and fileHandler() which also check for valid files and folders.
    - Upon each new directory or file, a new thread is created.
    - directoryHandler() returns the number of threads created.
    - The optional target input and output directories are specified via optional command line flags -d and -o flags (if unspecified, default parameters to sort in place are used).

- Sorting by column is specified with the flag -c
- Eg: ./multiThreadsorter -c <columnName> [-d <input-directory>] [-o <output-directory>]

HOW TO USE:
- To compile, open your terminal the directory holding "multiThreadsorter.c", "multiThreadsorter.h", "mergesort.c", "utilities.c", and "sorter.c" and the corresponding makefile.
- Run "make clean" (optional) and "make", then type "./multiThreadsorter -c <columnName> [-d <input-directory>] [-o <output-directory>]" where "filename" is the name of the CSV to be sorted, "columnName" is the column to be sorted by which should appear in the CSV, "-d" and "input-directory" to specify where to begin searching, and "-o" and "output-directory" to specify the output directory of the sorted CSV files.

Upon completion of the program, CSV(s) will be created with the content sorted as specified by the user, in the specified output folders if given.

ORGANIZATION:
- enum _format{ NUMBER=1, STRING=2 } format: Holds enum formats for easy comparison when determining a column's type to sort by.
- struct _Row: char** entries which holds the array of strings of entries for all the columns
- struct _Header: char** titles which holds the array of strings of the titles from row 0. format* type is an array which holds the enum format for the column
- mergesort(): calls merge() and sorts Rows based on specified data _format
- GetLine(): Reads in CSV files and parses rows into Row and Header Structs
- fileHandler(): Validates the input file and calls Sort() if properly formatted
- directoryHandler(): Validates directories and files and create a new thread upon new directories and files. Increment thread count
- Sort(): Mergesorts on given file descriptor, and writes to the output FD

ASSUMPTIONS:
- Must utilize stable sorting
- Create generalized version from the start to work with any CSV file.
- Some of the movie_metadata.csv entries had printed characters that we assumed were spaces but actually were not. We had to account for this special character in our code.
- Initially looking at the data, there are empty/null entries. We

made sure to sort these to the top as instructed.
    - If everything in a column is a number but meant to be a string (eg. movie names as only numbers), it should be sorted as an number anyway. If there is a majority numerical, but with strings, then they should be sorted as strings.
    - We assume we will only ints(+/-), doubles(+/-), or string and won't get pi, e, complex numbers, scientific notation.
    - Empty files
    - e and pi count as doubles
    - CSV file must be parsed and checked for the full name prior to the suffix ".csv".
    - Ignore sorted files with the structure <filename- -sorted-<column_name> .csv
    - Output directory doesn't require folder structure, simply the sorted files
    - Do not expect multiple files with the same names in any (sub) folder
    - Create new thread on any file, regardless if CSV
    - Regarding printing: only print out the pID stuff specified in the description, but print all ERRORS to STDERR (prevents IO stream collision)
    - pID output order does not matter
    - Filenames can have anything (eg. "somestuff.csv.txt.csv") so don't read until "csv", read until /0 and just go back 4 characters
    - Flag order doesn't matter. If flag not present use default behavior for it
      - -o = create sorted CSV in same folder as source CSV
      - -d = start searching the current directory
    - FATAL ERRORS
        - Errors with flags, in which case bail (eg. flag w/ no arg)
        - Malformed CSV's
            - Each row must have n cells where n = # of columns
            - Must terminate w/ newline
    - NOT ERRORS
        - CSV's with missing target column - just leave it alone
    - CSV file with column errors (count, missing sortby, etc.), print to STDERR
    - Each file and folder will be uniquely named

DIFFICULTIES:
    - Initial difficulties involved storing the entries into character arrays. The solution was to dynamically allocate space for singly and doubly pointed arrays to hold arrays of strings and arrays of row objects holding arrays of strings.
    - Uses read() in junction with open() to read from file vs stdin
    - Determining how to store the type of the column and storing column header/data. We decided to create a single column object which refers to each entry of the first row of the CSV. The struct we created would hold the string entry and the type via enum of

{ NUMBER=1, STRING=2 }

TESTING PROCEDURE:
    1) The testing procedure involved first printing out each line's string array after it was parsed and tokenized to ensure the file was read properly.
    2) Then, creating each row object and inserting them all into the Rows array, I printed out each one to ensure all the entries were properly separated and copied correctly.
    3) For merge sort, we hard coded test cases into row structs and sorted them.
    4) Created multiple multi-level (root to n subfolders) directories with CSV files, malformed CSV files, and non-CSV files