CST8234 – C Programming W18 (Lab 6)

Programming Exercise

In this lab, we will gain experience with

- Reading data from files, parsing it, then writing it out to another file

Statement of the problem

As with Lab 5, you are only required to implement one file in an otherwise functional program.

In this case, you are provided with a number of files, and it is your responsibility to process the data contained therein (where the filenames are passed into your program as command line arguments).

The files you will be reading will contain RentalProperty information that we've been using in the previous labs/assignments (e.g., they have an address, a rent, a number of rooms).

You have been given 5 data files. Two of them contain bad records that you are expected to detect and either warn about, or fail on. The files with bad records are called "properties_bad1.in" and "properties_bad2.in". The three remaining files are called "properties_low.in", "properties_med.in", and "properties_high.in". These all contain good information, and you should not generate any warnings or errors for the data contain within. For your information, the fundamental difference between the "low", "med" and "high" data files is solely that they contain properties in different price ranges.

Each row of each file represents one rental property. As you read each row, you will increase the size of a dynamically-resized array (i.e., using realloc) of RentalProperty structs. You'll populate the newly allocated struct in your array with the parameters contained in the file (e.g., streetName, streetNumber, rooms, rent).

Your array of rental properties will be terminated by a RentalProperty struct that has a negative rent. I.e., just like a "string" in C is terminated by a special character (i.e., a null), your list of properties will be terminated by a special rental property (i.e., with a negative rent). And, just like a string requires that you allocate space for the extra null character at the end of the string (i.e., "hello" requires 6 bytes), you will have to make sure that your array of rental properties has room for N+1 structs, since the very last one in the array will be the terminator.

As a convenience, the file rental.c has a method called 'countProperties()', which will tell you how many non-terminator properties are in a list.

Input Format

Each row in the input files can have the rental property values specified in any order, delimited by commas. Take a look at one of the good data files, for an example of how the values can be jumbled. You must parse each row, and deal with each value as you read it. You will find this process to be very similar to how you parsed a bunch of GNU long option command line args in Lab 5.

You need to detect that you've set all the values for each new RentalProperty struct. E.g., if a line is missing a one of the values (e.g., 'rooms') then you need to report the error and quit. If you find an extra value (e.g., 'distance') then you need to warn about the value, but continue processing. The two "bad" data files contain examples of poorly-specified records.

Command Line Arguments

You have been provided with a complete "parseArguments" function that will process files by calling a stub "readProperties" function (that you have to complete!), as well as an optional "--append" option that will be used when you output the sorted values by calling a stub "writeProperties" function (that you have to complete!)

If the "--append" (or "-a") option is not specified, then list of Rental Properties that you parse from the provided files will be sorted, and then written to a "properties.out".

If the "--append" (or "-a") is specified, then the list of Rental Properties that you parse from the provided files will be sorted, and *appended* to an existing "properties.out" file. (i.e., see the "fopen" modes).

The bottom line is that executing either of the following should generate the *identical* output.

```
./lab6.exe properties_med.in properties_high.in properties_low.in
```

```
./lab6.exe properties_low.in; ./lab6.exe -a properties_med.in; ./lab6.exe -a properties_high.in
```

Output Format

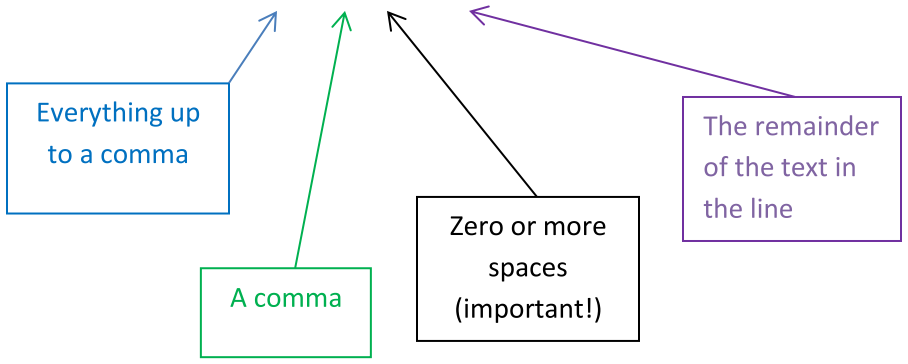
The output format should be compatible with the input format. I.e., if you take the resulting "properties.out" file, and re-run "./lab6.exe properties.out", it should successfully parse the data file without warnings or errors.

Parsing Hints

When processing the data in each file, I suggest you read in an entire line at once using the built-in 'fgets()' function. You will then end up with a string that you can parse (and re-parse) at your leisure using 'sscanf()' as discussed in lectures.

You may find the following instruction very useful for parsing the comma-delimited rows of data!

```
int rc = sscanf( strLine, "%[^,]%[,] %[^\n]", strArgument, &comma, strRemainder );
```


Everything up to a comma
A comma
Zero or more spaces (important!)
The remainder of the text in the line

Then, after each sscanf, if you end up with an rc > 1, you'll know you encountered a comma, so that you'll need to reprocess the remainder of the line for the next argument. If you have to reprocess the line, then keep in mind that you'll want to be calling sscanf on the remainder of the input… so you may want to copy the contents of strRemainder into strLine so you can just reprocess it in a loop.

It is permitted to define string buffers of some large size (e.g., 512 characters) for each for 'strLine', 'strArgument' and 'strRemainder'.

Requirements

1. Create a folder called algonquinUserID_L6 (e.g., "myna0123_L6"). Do all of your work in this folder, and when complete, submit the zipped folder as per the "Lab Instructions" posted on Blackboard.
2. Extract the files (.c, .h, makefile) in the lab attachment into your new folder. ***YOU ARE NOT REQUIRED TO MODIFY ANY OF THE CODE THAT YOU ARE BEING PROVIDED, except for file.c!***
3. Your job is to COMPLETE a file called file.c that defines two functions called `readProperties` and `writeProperties` that have already been declared in "file.h".
4. You are expected to read the code you have been provided, and understand how it runs. You may be tested in subsequent quiz on your understanding of the program's functionality.

Submission

When you are done, submit your program to Blackboard. Make sure that you have the appropriate header in your source file(s), and have zipped up the appropriately name directory. Only include the source code (.c, .h) ***including*** the files that were provided to you for this lab and your makefile (mandatory!). *Do not forget to zip up the directory… not just the files. I.e., when I open your zip file, I expect to find a folder called "myna0123_L6" that I can simply drag to a folder*. Do not include any other files (executables, stackdumps, vi "swap" files).

You ***must*** include a makefile, as part of your submission! When grading your reports I will unpack your zip file and type '`make`'… and if I don't end up with a 'lab5.exe' to run, ***I will not grade your assignment***.