CST8234 – C Programming W19 (Lab 5)

Programming Exercise

In this lab, we will gain experience with

- Parsing Command Line arguments
- Complying with long, detailed and boring problem requirements (i.e., read carefully!)

Statement of the problem

When you run a program from the command line, you usually have the ability to specify multiple arguments. For example, we can run the compiler by typing

```
gcc -g -o lab5 -ansi -pedantic lab5.c rental.c sort.c parse.c
```

Some of these arguments may consist of just the flag (e.g., "-g", "-ansi", etc.) and some of these may be a flag and value pair ("-o lab5"). The vast majority of arguments are optional, but it's not unusual to find programs that have a couple of mandatory options.

It's also typical that these can be provided in any order, e.g., in the gcc example above you could put "-o lab5" at the beginning of the argument list, or the end; and likewise you could drop "-g" in anywhere (except in the middle of "-o lab5").

There are two common conventions for specifying arguments, as illustrated by typing "diff --help"

```
% diff -help
Usage: diff [OPTION]... FILES
Compare FILES line by line.

Options
      --normal                  output a normal diff (the default)
  -q, --brief                   report only when files differ
  -s, --report-identical-files  report when two files are the same
  -c, -C NUM, --context[=NUM]   output NUM (default 3) lines of copied context
  -u, -U NUM, --unified[=NUM]   output NUM (default 3) lines of unified context
  -e, --ed                      output an ed script
```

I.e., there's the "POSIX" format of "-U 25" and "GNU long options" format of "--unified=25" (See https://www.gnu.org/software/libc/manual/html_node/Argument-Syntax.html). And many/most programs (like diff) support both for legacy reasons. I.e., you can type in either "diff -U 25" or "diff --unified=25" and it'll do exactly the same thing.

This lab is a little different in that you are being given 80% of the code, and are only responsible for one part… the processing of command line arguments.

Requirements

1. Create a folder that uses both members' Algonquin ID's (e.g., "smit9112_sing0003"). Do all of your work in this folder, and when complete, submit the zipped folder as per the "Lab Instructions" posted on Brightspace.
2. Extract the files (.c, .h, makefile) in the lab attachment into your new folder. These files implement program that lists rental properties. ***YOU ARE NOT REQUIRED, NOR PERMITTED, TO MODIFY ANY OF THE CODE THAT YOU ARE BEING PROVIDED, except for parse.c!***
3. Your job is to COMPLETE a file called parse.c that defines a function called `parseArguments` that has already been declared in "parse.h". As you will note, this function takes `argc` and `argv` (as passed into `main`), and bunch of pointers to variables that have been locally defined in main.c. The stub implementation of `parseArguments` simply prints out a sample usage message, but your implementation of `parseArguments` must implement logic that satisfies the following rules (read carefully!):
    a. Provides a usage message, similar to that which you see if you use the "-h" or "--help" arguments with most Linux/unix/Cygwin programs. You should be able to create a usage message that looks exactly like all the other messages you will see, including the usage text, description list of arguments, etc. I strongly suggest executing several programs ('diff', 'sort', 'uniq', 'du', 'df', 'ls', 'ps', etc.) with the "--help" option and you'll note there is a very consistent format. Formatting matters and marks will be deducted for spelling. A sample, but incomplete, function `printHelpAndExit()` has been provided for you to work from… but it's not entirely complete (you have to complete it).
    b. Accepts an argument that specifies the sorting method that is applied to the rental properties (e.g., by rent, by number of rooms, by address). This sorting has already been implemented for you… your responsibility is to simply get the user's choice and set the `eSortMethod` be one of the `SORT_METHOD` enum values, i.e., `sortByAddress` (ascending), `sortByRent` (ascending) , or `sortByRooms` (descending). **This flag/value argument is mandatory.** If the user provides a sort method that does not conform to one of the three ones you're expecting, you will print an error message.
    c. Accepts an argument that specifies the minimum value to print, given the selected sort method. E.g., a minimum number of rooms of 2, or a minimum rent of $800. If not provided, this parameter should default to zero. Specifying this value with the `sortByAddress` sort method is an error.
    d. Accepts an argument that specifies the maximum value to print, given the selected sort method. E.g., a maximum number of rooms of 3, or a maximum rent of $1000. If not provided, this parameter should default to a suitably large number (e.g., use `INT_MAX` as defined in limits.h). Specifying this value with the `sortByAddress` is an error.
    e. Accepts an argument that specifies the maximum number of rental properties to list.
    f. Accepts an argument (just a flag… no value) that specifies whether the properties are to be printed in reverse order, once the sorting has been applied. Note that the default sort order for `sortByRent` and `sortByAddress` is ascending, so the default value for `*pbReverse` should be false, unless the reverse flag is specified in which case

`*pbReverse` should be true. However, the default sort order for `sortByRooms` is descending, so the default value of *pbReverse should be true, and should only be false if the reverse flag is specified.

   g. Prints an error message if any unrecognized command is provided.
   h. Prints the usage message if no arguments are specified on the command line
   i. The arguments can be provided in any order. I.e., you may not make any assumptions that one flag must occur before another. If your implementation requires that the flags be in specific order, the best mark you will receive for the implementation is 30%.
   j. All flags and values are case sensitive. I.e., you are permitted to print an error message if the user enters a flag or keyword in the wrong case (e.g., “-R" instead of “-r”)
   k. Numeric arguments need to be correctly-specified numbers. E.g., your algorithm should print an error if you were expecting “-n NUM” and the user entered “-n x” or “-n 25a”.
   l. Any time an error is encounter in the arguments, e.g., an invalid/missing value, illegal argument combination (e.g., providing a min/max value with --sort=byAddress), you’ll print out the error and follow it with the usage message, and exit with the `EXIT_FAILURE` return code.
   m. Otherwise, you will exit with `EXIT_SUCCESS`.

4. You must support ***BOTH*** the POSIX or GNU long option conventions.
5. You must write your own solution, entirely. Yes, there is an awesome “getopt” library that helps you with much of the processing, that you’ll definitely want to use in the workplace. But the goal of this lab is to make you understand the processing that goes on behind the scenes when you use something like “getopt”. ***Consequently, if you use getopt, or copy its definition, you will receive ZERO on the implementation.***
6. You are expected to make your code as efficient as possible. If you find yourself copying and pasting code (e.g., all of min, max, and num results all take an integer value flag that needs to be validated), then try to put the common functionality into a function that can be called by all three arguments. Failure to do so may mean you lose marks for efficiency.
7. 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 Brightspace. 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 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***.