Project PL3

Due Date: October 4

Purpose

This project is to help you get familiar with the C language, and to experience why C is the language of choice for systems programs. In fact, Unix and Linux are written in C.

Problem

MacroStiff was very happy with your work on project PL2. Now they want tools that are even more useful by being operating system independent. To this end, they want you to write more system tools, but now written in C. Thus, your programs will run on any platform for which there is a C compiler. This means that you could theoretically use these Linux-like system commands on the command line in Windows!

The following is a list of tools they wish you to implement in C. As in project PL2, you will have multiple programs:

1. `Caverage` *filename* - find the average of a list of numbers stored in the file named *filename*. Same as in PL2, only this time written in C.
2. `Ccp` *filenameA filenameB* - copy the contents of *filenameA* to a file named *filenameB*; see Linux's *cp* command.
3. `Ccat` [OPTION] *filename* - display the contents of *filename*, where the options are (including none):
   - –n - number all output lines
   - –b - number nonempty output lines

   See Linux's *cat* command.
4. `Cwc` *filename* - find the number of lines, words, and bytes (characters) in *filename*; see the *wc* command.
5. `Csort` [OPTION] *filename* - sort the contents of *filename*, where the options are (including none):
   - –n - numeric sort
   - –r - reverse sort
   - –u - output only the first of an equal run of strings or numbers
   - –nr - numeric sort in reverse order

   See the *sort* command.

Input

For `Caverage`, the file contains a list of floating point numbers, one per line. There is no other data in the file.

For `Csort`, the file contains a list of strings/integers/floats, one item per line.

For the rest of the programs, the input file contains an unknown number of lines of ASCII text, which may or may not include numbers in both integer and floating point form.

If some error condition occurs, then the program's behavior should be consistent with Linux commands, as before.

Output

Output should be as defined for the problems above and/or how you think Linux would handle it. Following Linux command conventions, labels and explanatory text is **not** necessary. In any case, check out how all of the commands work in Linux with various input data.

Specifics

- C program file names should end with .c.
- All of your programs should be written in ANSI C. You can use any compiler/IDE on any platform, but be sure that you are using standard C (watch out for Microsoft products!). You may wish to test your program in csLab using the gnu C compiler. An example compile directive is as follows:

  ```
  gcc -ansi Ccw.c -o Ccw
  ```

  where

  - gcc = gnu c compiler
  - -ansi = ANSI flag which denotes standard C
  - Cmedian.c = file to compile
  - -o = compiler option that redirects output (executable) to a file
  - Ccw = name of the executable file. If you don't redirect the output, the default executable file name is *a.out*.
- Create a Makefile that will compile all of your programs and create the correctly named executable files. Write the file so that when I type:

  ```
  make all
  ```

  all of your programs will be compiled. Turn in the Makefile along with all of your source code.
- You should write **good** code; that is, break the program down into small functions. No function, including `main()`, should be longer than a printer page (about 60 lines).
- Be sure to name and implement all of your programs as described above. I will write my own script to test yours, and if something goes wrong because you have misnamed a file or put the arguments in an order not specified, your grade will suffer.

Notes

- As in the previous projects, include a comment at the top of each program with:
  - the name of the program
  - your name
  - a description of its function

  - a description of valid input
  - an exact description of what is output

- Comment each function, including each parameter.

- The programs get progressively more difficult. Don't wait to get started, as *sort* may take some time. Note that many of the programs will have similar input/output processing; it's OK to reuse your own code in each of the different programs.

- Write one program at a time, and add improvements incrementally. You should always have a set of programs that compile and run by the deadline, even if they don't do everything that is specified above; e.g., the `u` option doesn't work for `Csort`. It is better to turn in a partially working program(s) than a lot of code that doesn't run.

- Use established algorithms for common items, such as sorting. Copy a good sort function from a text or the web. Remember that it's **not** acceptable to copy code from a friend or to copy an entire program.

- As before, bundle all of your programs and Makefile into one tarball:

  `tar cvf` *yourLastName*`.tar` *file1 file2 file3 ...*

  Send the tar file to me via email before midnight on the due date. Hand in a printed copy of your programs in class the next day.

I must admit, there have been times in my life when I've felt calmer.
– Richard, on The Great British Baking Show (2015)