CST8234 – C Programming F17 (Lab 6)

Programming Exercise

In this lab, we will gain experience with

- Parsing Command Line arguments
- Complying with long detailed 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 lab6 -ansi -pedantic main.c parse.c deck.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 lab6"). 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 lab6" and the beginning of the argument list, or the end; and likewise you could drop "-g" in anywhere (except in the middle of "-o lab6").

Additionally, you'll find that most programs support a "-h" or "--help".

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 called algonquinUserID_L5 (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. *DON’T USE YOUR STUDENT NUMBER. I will deduct marks if you name your folder with student number or forget to include your user ID .*
2. Extract the files (.c, .h, makefile) in the lab attachment into your new folder. These files implement a more flexible version of the “card dealing” program we tackled earlier in the term. *YOU ARE NOT REQUIRED TO MODIFY ANY OF THE CODE THAT YOU ARE BEING PROVIDED!*
3. Your job is to create 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. 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. 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.
    b. Accepts an argument that specifies a numeric random number seed. This seed can allow the user to replay the same deal over and over again. If this flag/value pair is not defined, then the seed should default to “time(NULL)”.
    c. Accepts an argument that specifies whether a Euchre deck is to be used instead of a regular 52-card deck (a Euchre deck only has 24 cards in which the ranks 2-8 have been discarded). The implementation of the Euchre deck has already been done for you… you just need to allow the user to specify whether they want to use it or a standard deck.
    d. Accepts an argument that specifies the number of players. This is mandatory, unless the user has specified a Euchre deck. If using a Euchre deck the number of players is optional, but if specified it can only be 4.
    e. Accepts an argument that specifies the number of cards per player. This is mandatory, unless the user has specified a Euchre deck. If using a Euchre deck the number of cards per player is optional, but if specified it can only be 5.
    f. If not using a Euchre deck you must ensure the user’s choices of cards and players are valid. If you don’t there *will* be a segmentation fault in the code you’ve been given… preventing by checking the parameters in the parseArguments function is your responsibility.
    g. Accepts an argument that specifies the sorting method that is applied before printing out the players’ hands. The value for this function is a keyword that specifies either no sorting is to be done, or that the cards are to be sorted by rank (e.g., all the 2’s before all the 3’s), or by suit (i.e., all the clubs before all the diamonds). Again this sorting has

already been implemented for you… your responsibility is to get the user's choice and set the modeSort variable to be one of the SORT_NOSORT, SORT_BYRANK or SORT_BYSUIT constants. If this argument is not specified, then the sort algorithm should default to SORT_NOSORT, unless the Euchre deck is being used in which case it should default to SORT_BYSUIT. If the user provides a keyword that does not conform to one of the three ones you're expecting, you will print an error message.

   h. Accepts an argument (just a flag… no value) that specifies whether the cards are to be printed in reverse order, once the sorting has been applied. Your parser must set the modePrint flag to the PRINT_REVERSE constant if this flag is set, otherwise it defaults to PRINT_NORMAL.
   i. Prints an error message if any unrecognized command is provided.
   j. Prints the usage message if no arguments are specified on the command line
   k. The arguments can be provided in any order. I.e., you cannot make any assumptions that one flag must occur before another.
   l. 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.
   m. Numeric arguments need to be correctly-specified numbers. E.g., your algorithm should print an error if you were expecting "-U NUM" and the user entered "-U x" or "-U 25a".
   n. There is no need to support concatenating of flags… e.g., some programs that have arguments "-a" and "-b" and "-c" allow you to specify "-abc" as a shortcut for all three. A good example of this 'ps'. But you do not need to support this.
   o. Any time an error is encounter in the arguments, e.g., an invalid/missing value, illegal argument combination (e.g., zero/negative number of players), you'll print out the error and follow it with the usage message.
   p. You will exit with EXIT_FAILURE upon printing the usage message.

4. You are responsible for picking the flags you wish to use for each argument… e.g., whether the user must specify the number of players via "-P NUM" or "-n NUM" or "--players=NUM" etc. is completely your choice… but it must of course be documented in the usage report. You may choose to follow either the POSIX or GNU long option convention, but you have to use that convention for all your arguments.
5. 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.

Marking

This assignment is out of 40

- 10 for the quality and accuracy of the usage message, and helpful error messages. Full marks will entail having a usage message that clearly explains what the arguments do, and error messages that provide useful feedback to the user so that they can re-run the program again knowing what they did wrong the first time.
- 20 for coding correctness (i.e., correct results), i.e., implementing ALL of requirement #3 above. There are lots of rules covered in requirement s #3a-p. Read them carefully, and test your code thoroughly.
- 10 for clear comments and coding convention. For full marks, your code should be indistinguishable from the code found in main.c and deck.c.

You can also lose marks for incorrect submission (e.g., including unnecessary files in the zipped folder), compiler warnings, typographic errors (including in comments).

There will be a bonus 5 marks if you support ***both*** "POSIX" convention (e.g., "-U 25") ***and*** "GNU long option" convention ("--unified=25"), for each requirement. But as with previous assessments, you are strongly encouraged to get the basic functionality working perfectly, before attempting the "stretch goals" in a copy of your finished file. Be warned… while it may seem like a simple task of comparing the flag against two arguments, e.g., ' if (strcmp("-u", strFlag) == 0 || strncmp("--unified=", strFlag, 11) == 0)' you'll actually have to process the value differently because in the POSIX format a flags value is provided as the next argument, and in GNU the value part of the same argument as the flag ("--unified=25"). And, if you implement support for both conventions by just copying and pasting common bits of logic over and over, I'll start to deduct marks for efficiency… so supporting both POSIX and GNU long format comes with the expectation is that you'll be clever about how you commonly test for and retrieve the values for flags.

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 'lab6.exe' to run, ***I will not grade your assignment***.