Introduction to C Programming in a Unix (Linux 32 bits) Environment

This lab assignment is to be done SOLO!

Home-Lab A -- Assignment goals:

- C primer
- Parsing command-line arguments
- Undestanding character encoding (ASCII)
- Implementing a debug mode for your program
- Introduction to standard streams (stdin, stdout, stderr)
- Simple stream IO library functions

Preparation - Part 0: Maintaining a project using make

You should perform this part **before** starting to implement the hand-in assignment. It is basic knowledge that is not checked, but needed to complete the assignment.

For this part, 3 files are provided: **add.s**, **main.c**, **numbers.c**. The first file is assembly language code, and the other 2 are C source code.

1. Log in to Linux.
2. Decide on an ASCII text editor of your choice (vi, emacs, kate, pico, nano, femto, or whatever). It is **your responsibility** to know how to operate the text editor well enough for all tasks in all labs.
3. Using the text editor that you decided to use, write a makefile for the given files (as explained in the introduction to GNU Make Manual, see the **Reading Material for Lab A** . The Makefile should provide targets for compiling the program and cleaning up the working directory.
4. Compile the project by executing make in the console.
5. Read all of lab A reading material, and make sure you **understand** it.
6. Read the puts(3) and printf(3) manuals. What is the difference between the functions? To read the manuals type man followed by the function name (e.g. `man puts`) in a "console".

Important

To protect your files from being viewed or copied by other people, thereby possibly earning you a disciplinary hearing, employ the Linux permission system by running: `chmod 700 -R ~` In order to make sure you have sufficient space in your workspace, run the following command once you're logged in `du -a | sort -n` Then you can see a list of your files/directories and the amount of space each file/directory takes. If you need space and KNOW which files to remove, you can do that by: `rm -f [filename]`

Unix: Control+D, Control+C and Control+Z

- What does Control+D (^D) do? Control+D causes the Unix terminal driver to signal the EOF condition to the process running in this terminal foreground. You can read more about it here.
- What does Control+C (^C) do? Pressing Control+C in the terminal, causes the Unix terminal to send the SIGINT signal to the process running in the terminal foreground. This will usually terminate the process.
- What does Control+Z (^Z) do? Pressing Control+Z in the terminal, causes the Unix terminal to send the SIGTSTP signal to the process running in the terminal foreground. This will suspend the process (meaning the process will still live in background).

- Do not use Control+Z for terminating processes!!!

Writing a simple program

First, re-read and understand the arguments of main(argc, argv), which represent the command-line arguments in the line used to run any program using a "console". Recall that argc is the number of arguments, and that argv is an array of pointers to locations containing "null terminated strings" - the command line arguments, with argv[0] pointing to the program file name used in the command line to run the program. Then, write a simple echo program named my_echo:
NAME
my_echo
SYNOPSIS - echoes text.
my_echo
DESCRIPTION
my_echo prints out the text given in the command line by the user (all whitespace printed as a single space).
EXAMPLES

```
#> my_echo aa b c
aa b c
```

Mandatory requirements

- Create a proper makefile as described in the reading material.
- Test your program to see that it actually works.

The actual assignment

Make sure you have done part 0 before you start implementing this assignment, especially understanding the semantics of arguments of main( ). In this simple assignment you will be writing a simple **encoder program**. The program has three functionalities:

1. Parsing the command-line arguments and printing debug messages.
2. The actual encoder.
3. Redirecting the input and output according to the command-line arguments.

Although you will be submitting a single program containing all the above, it is highly recommended that you implement each step in the above order and test it thoroughly before proceeding to the next one. There are several reasons for this. First, the step-by-step scheme is how physical labs will be run. But more in general, it is important to be able to partition the work and test functionalities separately, this leads to much more efficient and correct code development.

Part 1: Command-Line Arguments, Debugging, and input "echoing"

First, make sure you have done part 0 before you start implementing this assignment, especially w.r.t. understanding the semantics of arguments of main( ).

Second, introduce a debug mode into your program. For this we will develop an easy debugging scheme which can be used with any program and allows for special debugging features for testing. The minimum implementation prints out important information to stderr when in debug mode. Printing out the command-line arguments allows for easy detection of errors in retrieving them. Henceforth, code you write in most labs and assignments will also require adding a debug mode, and it is a good idea to have this option in **all** programs you write, even if **not required** to do so!

For this scheme, you must simply loop over the command-line arguments, and if in debug mode, print each argument on a separate "line" to stderr. Debug mode is a variable that you can choose to initialize to "on" or "off" (default: **on**), but if there is a command line argument "-D" it turns debug mode off, and if there is a command-line argument "+Dpassword" it turns the debug mode on. Note that here password is not meant to be the constant string "password", rather it should be a globally defined string like `unsigned char password[] = "my_password1";` which should be compared to whatever comes immediately after the "+D". For simplicity, we require the effect of a debug flag change to occur immediately after the current command line argument is handled, that is, starting with the **next** command line argument.

Use fprintf( ) -- see manual -- for simple printing of "strings" on separate lines. Note, that the output should be to stderr, rather than stdout, to differentiate between regular program output (currently null) and debug/error messages from the program.

After handling the command line arguments, the program should behave as follows, every input character c (from stdin) is simply sent immediately to the output (stdout). That is, you read a character using fgetc( ), pass it to a function called encode(c) that at present returns it unchanged (i.e simply do: c=encode(c), where encode just returns its argument for now) and then print c using fputc( ), until detecting an EOF condition in the input (preferably using feof( )), at which point you should close the output stream and exit "normally". We recommend here that you use global variables such as infile and outfile as arguments to fgetc() and fputc() respectively, initialized by default to stdin and stdout, respectively. This will allow you to do part 3 later on with very little effort.

Part 2: The Encoder

In this part you will first use the command-line parsing to detect a possible encoding string, and use that to modify the behaviour of encode( ). With no encoding string (default), every input character (from stdin) is simply sent to the output (stdout).

The encoding works as follows. The encryption key is of the following structure: +E{key}. The argument {key} stands for a sequence of digits whose value will be **added** to each input characters in sequence, in a **cyclic** manner.
This means that each digit in sequence received by the encoder is added to the corresponding character in sequence in the key. When and if the end of the key is reached, re-start reading encoding digits from the beginning of the key. You should support both addition and subtraction, +E{key} is for addition and -E{key} is for subtraction.

Implementation is as follows. The key value, if any, is provided as a command-line argument. As stated above, this is indicated by a command line argument such as "+E1234" or "-E13061". The first is a sequence of numbers to be **added** to the input characters before in encode( ), while the second is a sequence of numbers to be **subtracted** from the input characters. Assumptions are: only at most one of "+E" or "-E" are present, and the rest of the command line argument is always (only) a non-empty sequence of decimal digits, terminated as usual by a null character.

Encoding is as follows: to the first character of the input, add the numerical value of the first encoding digit, to the second input character add the (numerical value of the) second digit, etc. If you reach the end of the encoding string (null character!) before you reach EOF in the input, reset to the beginning of the encoding string. Observe that this is ASCII encoding, so it should be **very simple** to compute the numeric value of each digit, which you should do directly using no special functions.

Note that we advance in the encoding key once for each input character, but encoding, if indicated, should only be applied to upper case and digit characters, that is 0-9, and A-Z, and should use "wrap around", that is assume Z+1 is A, and A-1 is Z. etc. Examples are provided below to fully clarify this.

Examples
In the first example below see how the A,B,C,D,E are encoded adding 1,2,3,4,5 respectively and then for the next character the encoding key is reset to 1 for the next character, Z. But Z+1 is wrapped around and becomes A. Then there is a newline character, which is output with no change (still advancing the encoding key) so to the next character 3 is added, and to the one after that 4 is added. The last 2 characters again are output with no change. The 2nd example below is similar but now with a key to subtract, rather than add. In the following example, output to stderr is in **bold**.

```
#> encoder +E12345
encoder
+E12345
ABCDEZ
BDFHJA
12#<
```

```
46#<
^D
#> encoder -E4321
encoder
-E4321
GDUQP523
CASPL202
^D
```

Implementation note: it is a good idea to make the encoding string a global variable, and also to provide a default initial value of "0", which results in idempotent (i.e. no change of value) encoding. This way you do not need a special case for when no encoding is required according to the command line arguments!

Part 3: Input and/or Output to Specific Files

The default operation of your program is to read characters from stdin (the "console" keyboard), encode them as needed, and output the results to stdout (the "console" display). After checking correctness of all the previous parts, now add the option for reading the input from a specified input file: if command-line argument "-ifname" is present, the input should be read from a file called "fname" instead of stdin (or in general the file name starts immediately after the "-i" and ends at the null character). Likewise, if command-line argument "-ofname" is present, the output should go to a file name "fname" (or in general, file name immediately after the "o").

Observe that if you did things right and heeded our advice above, this part is only a few lines of code: while scanning the command-line arguments simply check for "-i" and "-o", open input and/or output files as needed using fopen( ), and use the file descriptor it returns for the value of "infile" and/or "outfile". The rest of the program does not need to change at all. Just make sure that if fopen( ) fails, print an error message to stderr and exit. Note that your program should support encoding keys, input file setting, output file setting, and debug flag setting, in any combination or order. You may assume that at most one of each will be given (e.g. no more than one encoding key, no more than one input file, and no more than one output file setting).

Optionally, when debug mode is on, your encoder may print to stderr any information you wish, such as the input character and the encoding value, to help you in debugging your code. It is OK to leave this in the submitted version, as we are not requiring a specific result in stderr.

Helpful Information and Hints

- stdin and stdout are FILE* constants that can be used with fgetc and fputc.
- Make sure you know how to recognize end of file (*EOF*).
- Control-D in an empty input line causes the Unix terminal driver to signal the EOF condition to the process running in this terminal foreground, using this key combination (shown in the above example as ^D) will cause *fgetc()* to return an *EOF* constant and in response your program should terminate itself "normally".
- Refer to ASCII table for more information on characters encoding.
- Remember that "character strings" in C are simply null terminated arrays of "unsigned char", to which, obviously, a pointer can point. Thus, when accessing parts of such strings it is better and simpler **not** to use library string functions such as strcpy, strlen, and the like. In fact, even though we allow you to use "strncmp" or "strcmp" to detect things like "+E" in the command line, a more concise and preferred implementation actually does **not** use them, instead comparing individual characters!
- In every do-at-home lab, we designate some issues as "may obtain help". By this we mean there will be an **optional** task in a concurrent physical-attendance lab (in this case lab 1), where you may obtain help from a TA and make sure you are doing things correctly. This does **not** mean that you can expect the lab TA to do your work for you, only help to clarify misconceptions and difficulties. In this assignment the following issues are designated as such:
    1. Correctly detecting the end-of-file condition in the input file.
    2. Correct use of makefiles.

Mandatory requirements

- You **must** read and process the input **character by character**, there is no need to store the characters you read at all, except for the character currently being processed.
- Important - you cannot make any assumption about the line lengths.
- Check whether a character is lowercase (letter resp. uppercase, or number) by using a single "if" statement with two conditions. How?
- You are **not** allowed to use any library function or macros for the purpose of recognizing whether a character is a letter, and its case.
- Points will be deducted for using string library functions "strlen", "strcpy", and all others except for "strncmp" and "strcmp". The latter (strncmp, strcmp) are allowed with no penalty, but not recommended.
- Read your program parameters in the same way as in Part 0's: main.c. First, set default values to the variables holding the program configuration and then scan through *argv* to update those values. Points will be reduced for failing to do so.
- Program arguments may arrive in an **arbitrary** order. Your program must support this feature.
- Output to stdout (or ofname) must contain **only** the (possibly encoded) characters from the input, this will undergo automated checking so any deviation will cause faling automated tests and a grade reduction. For the debug printouts (in stderr) we will not enforce a specific format.

Submission

In the following submission instructions and deliverables as well as point distribution.

Submission instructions

- Create a zip file with the relevant files (only).
- Upload zip file to the submission system.
- Download the zip file from the submission system and extract its content to an empty folder.
- Compile and test the code to make sure that it still works.
- In this case the makefile should be set so that "make encoder" generates an executable file of your program with the name "encoder" in its current directory, and with no additional directory structure.

Deliverables

A zip file containing exactly the following files:

1. makefile
2. encoder.c

Credit Points per Part

Part	Points
1	30
2	40
3	30