ECE 2220: System Programming Concepts **Fall 2016**
Machine Problem 4 **Due: 11:59 pm, Monday, October 10**

The goal of this machine problem is to learn to use dynamic memory and pointer arithmetic to build a simple file editor. Template files are provided to illustrate some of the basic concepts required for this program. In this lab, each student is to update the template called `lab4.c` to complete a simple file editor that can edit bytes in a file. The bytes do not need to be printable. The editor loads the contents of a file into dynamic memory and allows the user to search the contents for bytes or strings, replace the bytes or strings, and to move to specific locations in the memory.

Input

A template program is provided that accepts a command line argument with the name of a file to load into memory so its contents can be searched. The file should be read into memory by storing it in a dynamically allocated memory location that takes up the same amount of memory as the file.

A menu is displayed which has the following options:

`f FF`	Search for next matching byte (in hex) after the current location
`r AB CD`	Same as search, and then replace the byte **`0xAB`** with **`0xCD`** if found
`/Blue Ridge`	Search for next matching string after the current location
`s /Blue/Red /`	Same as search, and then replace with a string of the same length
`G num`	Goto byte at position 'num' from the start of memory
`j`	Move forward 16 bytes from the current location
`k`	Move backward 16 bytes from the current location
`q`	Quit

The template code that is provided collects and parses the input. Stub functions are provided that must be used to perform the required operations. Additional functions can be added, but no changes are permitted to the code in **`main()`**. Notice that most of the commands depend on the location of the "cursor", and the cursor position can change as a result of the operation. After the file is read into memory, the cursor position is initialized to the first byte of the file stored in memory. The **`f`**, **`r`**, **`/`**, and **`s`** commands begin the search at the first byte **after** the current cursor location. If a match is found, the cursor position is updated to the byte at the beginning of the match. If a match is not found, the cursor position is not changed. When searching for a byte or string, find the first occurrence. If the end of the memory block is reached before a match is found, wrap around to the first memory location and continue the search. The **`r`** and **`s`** commands change the memory block. Simply replace the first byte (string) found with the byte (string) given with the command, respectively. Note that for the **`s`** command the search string and the replacement string must be the same length. The **`G`**, **`j`**, and **`k`** commands simply update the location of the cursor. These commands are so simple that the code to update the location is already given in **`main()`** and no changes are needed.

For the search string with the **`/`** and **`s`** commands, one wildcard position can be optionally specified. The wildcard position is denoted with the first occurrence of the symbol `'.'`, and permits that byte within the search string to match any byte in the memory block. All other characters in a search string must be identical to the bytes in the memory block for a valid match. A wildcard `'.'` character (and multiple escape `'\'` characters) can appear in both search strings for the **`/`** and **`s`** commands. See the comments in **`lab4.c`** for the **`process_search_string`** and **`process_replace_string`** functions for additional details.

Output

Any command that changes the position of the cursor (**f**, **G**, **j**, **k**), or changes a byte in the memory block (**r**) must be followed by a print of the sixteen bytes of memory which include the position of the cursor. (There are additional requirements for the **/** and **s** command that are described below.) The block of sixteen bytes must be aligned with a starting offset such that the offset modulo 16 is zero.

The first row of the output is simply the last digit of the offset from the base address of the line as a decimal. The second row begins with the offset from the beginning of the memory block inside a `'['` and a `']'`, followed by the ASCII representation of the bytes where possible. If a character is not a printable ASCII character, a blank space must be displayed. The field width for the offset value must be 6. In the third row, print the hexadecimal representation of the contents at the memory location. Finally, in the fourth row, a caret (**^**) must be displayed at the location of the cursor.

For example, if the command **f 4C** ('**L**' - a.k.a. 4C in hex) is searched for and is located at an address that is offset from the start of the memory block by 1 byte, the output should show the following 16 bytes:

```
          0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
[     0]  C  L  E  M  S  O  N     T  I  G  E  R  S  !
          43 4C 45 4D 53 4F 4E 20 54 49 47 45 52 53 21 0A
             ^
```

If the command **/Where the Blue** is entered, the output is the following. In this example, the first letter in the string is found at address 1071. For strings, print each block of 16 bytes that contains part of the string; in this example the string is located across 2 16-byte words. In addition to marking the start of the string with a '**^**', mark the last location of the string with a '**|**' and all the characters in the interior of the string with '**-**'. A string can be up to **120** characters. Mark the position of a wildcard with a '*****'.

```
> /Where the Blue Ridge yaw

          0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
[  1440]  _     b  p     N  W  h  e  r  e     t  h  e
          5F AE 62 70 8A 4E 57 68 65 72 65 20 74 68 65 20
                            ^ --------------------------

          0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
[  1456]  B  l  u  e     R  i  d  g  e     y  a  w  n  s
          42 6C 75 65 20 52 69 64 67 65 20 79 61 77 6E 73
         -----------------------------------------|

> /Clems.n

          0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
[   256]                                         C  l  e
          F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE 43 6C 65
                                                 ^ ------

          0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
[   272]  m  s  o  n     T  i  g  e  r  s  !
          6D 73 6F 6E 20 54 69 67 65 72 73 21 0A A3 97 A2
         -------*---|
```

Your program will be tested on a file supplied by the instructor so make sure your testing incorporates various possibilities to insure its proper operation. In particular, some bytes will contain values that are not printable characters.

It is critical that the output be formatted exactly as shown in the above figures including printing the addresses as decimal numbers. In particular the field width for the starting memory address for a 16-byte word must be exactly 6 characters.

Three example programs are provided in addition to the lab template: **`dyn1.c`**, **`filesize.c`**, and **`ptrprint.c`**. The first program, **`dyn1.c`**, illustrates use of dynamic memory and pointer arithmetic. The program **`filesize.c`** demonstrates how to open a file and store it in dynamic memory. Finally, **`ptrprint.c`** shows how to use pointer offsets to print a memory location and mark a selected item. The techniques illustrated in these programs will be useful in completing this assignment.

Notes

1. You compile your code using: `gcc -Wall -g lab4.c -o lab4`

The code you submit must compile using the `-Wall` flag and **<u>no</u>** compiler errors <u>or</u> warnings should be printed. To receive credit for this assignment your code must load in a file and at a minimum find and print one match for each of the options. Code that does not compile or fails to pass the minimum tests will not be accepted or graded.

2. Submit your file `lab4.c` to the ECE assign server. You submit by email to ece_assign@clemson.edu. Use as subject header ECE222-1,#4. When you submit to the assign server, verify that you get an automatically generated confirmation email within a few minutes. If you don't get a confirmation email, your submission was not successful. You must include your file as an attachment. You can make more than one submission but we will only grade the final submission.

See the ECE 222 Programming Guide for additional requirements that apply to all programming assignments.

Work must be completed by each individual student. See the course syllabus for additional policies.