ECE 2220: System Programming Concepts **Fall 2020**
Machine Problem 4 **Due: 11:59 pm, Saturday, October 24**

The goal of this machine problem is to learn to use dynamic memory and pointer arithmetic using a packet sniffer as a framework. 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 packet sniffer that can find relevant data within packets. The packets will be comprised of bytes of data – these bytes do not need to be printable. The input packets will be stored in a file. The packet sniffer loads the contents of the file into dynamic memory and allows the user to search individual packets for bytes and strings, search all of the packets for bytes and strings, search specifically for website domain names, or print a specific packet.

Input

A template program is provided that accepts a command line argument with the name of a file that contains packets to load into memory so their contents can be searched. The file is read into memory by packing each packet into a dynamically allocated memory location.

A menu is displayed which has the following options:

`f FF num`	Search for first instance of the provided byte (in hex) in packet number num
`g FF`	Search for first instance of provided byte in each packet
`s string num`	Search for first instance of provided string in packet number num
`t string`	Search for first instance of provided string in each packet
`w`	Search all packets for website domain names
`p num`	Print packet number num (formatted print)
`q`	Quit
`?`	Print this menu

The template code that is provided collects and partially processes 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()`**.

When the program begins, **`main()`** calls malloc() and dynamically allocates a large space named *packetspace*. It also creates an array of pointers named *packets*. It then calls the pre-written function **`process_input()`**. **`process_input()`** opens the file specified on the command line and extracts the packets from it, one at a time. Each time it pulls a packet, **`process_input()`** calls the to-be-written function **`store_packet()`**. **`store_packet()`** accepts the dynamically allocated space *packetspace*, the array of pointers *packets*, the packet to-be-stored named *packet*, the *length* of that packet, and the packet's number - named *packetnum*. **`store_packet()`** takes the provided *packet* and stores it within *packetspace*. At the same time, it updates that packet's entry within the pointer array *packets* so that *packets*[*packetnum*] contains a pointer to the start of that packet. It also updates the value at *packets*[*packetnum*+1] with a pointer to where the next packet will start.

After the packets are stored into dynamic memory, the above menu is displayed. Each menu selection will call an associated function (e.g., **`g FF`** will call **`find_byte_all()`**). Stubs for the functions called by the commands are provided within the template file. Detailed descriptions of function behavior can be found in the header comments of these stubs. (Note that the functionality for commands **`q`** and **`?`** has already been provided in **`main()`**.)

Output

If commands **f** and **s** are successful, the entire packet is printed - with the byte/string indicated.
If commands **g** and **t** are successful, each packet with a match is printed - with the byte/string indicated.
If command **w** is successful, a list of found websites is printed, including duplicates.
Command **p** always prints the specified packet - with no additional indication.
Note that packets are numbered starting at 0, so that the first packet is packet 0.

To print a packet, you must write and call the function **print_packet()**. This function creates a formatted print that displays the packet sixteen bytes at a time.

The first row of the output is simply the last digit of a count from 0 to 15. The second row begins with an indication of which piece of which packet this line is coming from, stored inside a '[' and a ']'. For example, **[3-4]** indicates packet 3, piece 4. The rest of the second line is the ASCII representation of the bytes where possible. If a character is not a printable ASCII character, a blank space must be displayed instead. The field width for the packet number must be 4, and the field width for the packet piece number must be 1. 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 point of interest.

For example, if the command **f 61 4** ('**a**' - a.k.a. 61 in hex) is entered, then packet 4 is searched for the byte value **61**. If the first 'a' is located at an address that is offset from the start of the packet by 5 bytes, the output should show the following 16 bytes:

```
          0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
[   4-0]  P  r  i  s  m  a  t  i  c
          50 72 69 73 6D 61 74 69 63 XX XX XX XX XX XX XX
                         ^
```

Here, the entirety of the packet is just the word "Prismatic". Notice that bytes not contained within the packet are marked with XX as the hex value. Also notice that the packet piece number starts with 0 and counts up.

Now consider the command **s phenomenon 0** for which the output is the following. In this example, the first letter in the string is found at position 28: packet piece 1, position 12. For strings, mark 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 could comprise the entirety of the packet.

```
          0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
[   0-0]  W  e  l  l  ,     a     d  o  u  b  l  e     r
          57 65 6C 6C 2C 20 61 20 64 6F 75 62 6C 65 20 72

          0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
[   0-1]  a  i  n  b  o  w     i  s     a     p  h  e  n
          61 69 6E 62 6F 77 20 69 73 20 61 20 70 68 65 6E
                                              ^ ---------
```

```
           0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
[   0-2]   o  m  e  n  o  n     o  f     o  p  t  i  c  s
           6F 6D 65 6E 6F 6E 20 6F 66 20 6F 70 74 69 63 73
          -----------------|

           0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
[   0-3]   .  .  .
           2E 2E 2E XX XX XX XX XX XX XX XX XX XX XX XX XX
```

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. Note that some packets may contain bytes such as '\n' and '\0' that may erroneously manipulate string functions.

It is critical that the output be formatted exactly as shown in the above figures including printing the first line as decimal numbers. In particular, the field width for the packet number and packet piece must be exactly 6 characters.

We will provide example input and output files, and your output files from the tests we provide must match exactly. Note that during grading we will use additional tests beyond those supplies. Verify that your code works with the provided files using either **`diff`** or **`meld`**, such as was done for MP2.

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 **no** compiler errors or 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 Canvas.

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.