Objectives

1. To get deeper understanding on memory addresses and C pointers.
2. To learn how to use pointers in C programs
3. To learn how to handle strings, array of pointers, and pointers pointing to strings in C programs.
4. To understand the structure of argv command line parameters and learn how to parse command line parameters using getopt().
5. To learn how to process environment variables.

Problem 1

Write a C program that "examines" the memory storing the input parameters of the *main()* function (i.e., argv and all command line parameters). The program "examines" the memory in a similar way as what the "*examine"* gdb command does with the "xb" format, i.e., taking memory space as a bit string and printing out the bytes in this bit string in a hexadecimal format. But your program only "examines" the memory space containing the pointers and strings related to argv. They include the argv variable (first part in the sample output below), the array of pointers (second part in the sample output), and the command line parameters (the third part in the sample output).

Assume the executable file of the program is named *examine*. Executing command "*./examine this is CS 288*" prints the following output. Do not use the example addresses below. Obtain your own memory addresses and draw them by executing your own program. The output of your program does not have to as the same as below, but somehow you need to be able to convey the memory addresses and data.

```
argv     | 00 00 7f fd de 8c fd a8 | 0x7ffdde8cfc90

argv[0] | 00 00 7f fd de 8d 07 25 | 0x7ffdde8cfda8
argv[1] | 00 00 7f fd de 8d 07 2f | 0x7ffdde8cfdb0
argv[2] | 00 00 7f fd de 8d 07 34 | 0x7ffdde8cfdb8
argv[3] | 00 00 7f fd de 8d 07 37 | 0x7ffdde8cfdc0
argv[4] | 00 00 7f fd de 8d 07 3a | 0x7ffdde8cfdc8

         |65(e)  2f(/)   2e(.)   00(\0)  00(\0)  00(\0)  00(\0)  00(\0)  |0x7ffdde8d0720
         |74(t)  00(\0)  65(e)   6e(n)   69(i)   6d(m)   61(a)   78(x)   |0x7ffdde8d0728
         |43(C)  00(\0)  73(s)   69(i)   00(\0)  73(s)   69(i)   68(h)   |0x7ffdde8d0730
         |53(S)  4c(L)   00(\0)  38(8)   38(8)   32(2)   00(\0)  53(S)   |0x7ffdde8d0738
```

In the above output, each line shows 8 bytes of memory contents, with the most significant byte (MSB) on the left and least significant byte (LSB) on the right. The order is important to show the memory contents in pointers. For example, the memory address in argv+0 is shown as `00 00 7f fd de 8c 07 25` with MSB on the left, but it would become `25 07 8c de fd 7f 00 00` with MSB shown on the right (not friendly to read).

On each line, the starting memory address is shown on the right. For example, character `x` is saved at memory address `0x7ffdde8d0728`. It is desirable to have the starting memory address **8 byte aligned** (last hexadecimal digit is either 0 or 8). You may use *%p* in printf. For example, *printf("%p", &a)* prints out the memory address that saves variable *a*.

Since each line contains 8 bytes and the data related with argv is not always 8 byte aligned, it is possible that the line shows some bytes not related to argv. For example, the first command line parameter starts at

address `0x7ffdde8d0725`, but the corresponding line starts at address `0x7ffdde8d0720`. So you can find 5 bytes printed on the first line (i.e., the '\0's). On the last line, you can also find two extra characters ('L' and 'S') after the NULL character of the last command line parameter.

For the memory addresses saving pointers (e.g., argv, argv[0], argv[1], etc), the pointers are shown on the left as labels. The memory contents are shown in hexadecimal format. You may use %02hhx in printf to print a byte (char) in two hexadecimal digits, e.g., *printf("%02hhx", c)*.

For the memory addresses saving command line parameters, in addition to hexadecimal values, your program needs to show the corresponding characters. Show escape sequences if the characters are not printable, e.g., '\0'. (Use *"\\%d"* in printf.) Use function *isprint()* to determine whether a character is printable or not.

When writing your program, don't directly access the data using the argv pointers (e.g., argv[0]). An easier way is to do the following. Extract a starting address that need to be examined, save it into an `unsigned char *` pointer. So you can use the pointer to access the byte and move the pointer (++) to get other bytes. For example, to print out the third part, save argv[0] (i.e., memory address `0x7ffdde8d0725`) into a pointer, move the pointer to the closest lower address, which is a multiple 8 (i.e., memory address `0x7ffdde8d0720`), starting from here, in a loop, print out the byte pointed by the pointer, and move the pointer, and print out the byte and move the pointer again, …

To determine whether the output of your program is correct or not, check the pointers (addresses) in the output: based on the memory addresses in the pointers, find in the output the corresponding data pointed by the pointers; then, determine whether it is the data that should be pointed by the pointers. For example, in the sample output above, argv[0] is *00 00 7f fd de 8d 07 25*; corresponding to this memory addresses, in the output we can find all the characters in argv[0] (i.e., ./examine) and '\0'. We can repeat the examination for argv[1]~argv[4], and conclude that the output is correct.

Problem 2

Write a C program that uses `getopt()` to parse its command line. Refer to the example program for `getopt()` in the slides. Assume that you were writing a program named `my_uniq`, which accepts the same command line parameters as the *uniq* tool does in Linux:

```
my_uniq [-c|-d|-u] [-f fields] [-s char] [input_file [output_file]]
```

Note that your program only extracts and prints out the options, option parameters, and other arguments in the command line (similar to the example program for `getopt`() in the slides). Your program does not need to determine whether the command line is valid or not. But, if there are invalid options and/or option parameters, it needs to print out error messages. It does not check whether the operands are really files, does not open/process the input file, and does not generate any outputs other than the lines reporting the command line arguments and/or error massages.

Problem 3

Write a C program that sorts all environment variables based on their names. Environment variables are obtained with the *envp* parameter of the main() function. The lecture slides have shown a program printing out all the environment variables. You may refer to the program for how to access environment variables.

Each environment variable has a variable name (the part before the '=' sign) and a variable value (the part after the '=' sign). For the following two entries in *envp* (i.e., two environment variables), their names are *USER* and *PWD*, respectively, and their values are *ubuntu* and */tmp*.

```
"USER=ubuntu"
"PWD=/tmp"
```

Your program needs to sort the environment variables based on their names. It can use *strtok()* to get the environment variable names. To sort the environment variables, it can exchange the pointers saved in the array pointed by *envp*. It can use bubble sort (https://en.wikipedia.org/wiki/Bubble_sort). So, it does not need to create other data structures.

The environment variables are sorted in **ascending order** determined by *strcmp()* of their names. For example, to compare the above two environment variables, the program call *strcmp("USER", "PWD")*. Since *strcmp* returns an integer greater than 0, *"USER"* is greater than *"USER"*. Thus, "USER=ubuntu" should be put after "PWD=/tmp", i.e.,

```
PWD=/tmp
USER=ubuntu
```