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 in hexadecimal format. There are three parts. The first part (first line) shows in the middle of the line the 8B contents saved in variable argv, which is the starting address of the array of pointers (argv[0], argv[1], …). The line also shows the variable name on the left before '|', and shows the address of the variable on the right after '|". After an empty line, the second part shows the contents in argv[0], argv[1], etc, which are the memory addresses of the strings containing command line arguments. Also, each line shows variable names on the left and addresses of argv[0], argv[1], etc on the right. The third part is mainly to show the strings containing command line arguments. However, it contains all memory contents arounds these strings to give you a better picture on how these strings are saved in memory. For this, this part starts at a memory address that is aligned to 8B and ends at a memory address aligned to 8B. To contain all the command line arguments, this part starts before the first byte in first command line argument and ends after the last byte (i.e., the NULL character) of the last command line argument. The NULL characters at the end of other command line arguments are also shown in this part.

One each line, the most significant byte (MSB) is shown on the left, and the least significant byte (LSB) on the right. The order is important to show the memory contents in pointers. For example, with MSB on the left, the memory address in argv is shown as `00 00 7f fd de 8c fd a8`, which is consistent with the address shown on the first line of the second part. If the MSB is shown on the right, the address will look like `a8 fd 8c de fd 7f 00 00`, making it difficult to compare with the address in the second part.

On each line, the **starting** memory address of the line 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, e.g., *`unsigned char *p=&argv`*. So you can use the pointer and offsets to access other bytes, e.g., *`for (i=7; i>=0; i--) printf("%02hhx", *(p+i);`* This method is particularly useful when printing out the third part. You may save argv[0] (i.e., memory address `0x7ffdde8d0725`) into a pointer, and move the pointer to the closest lower address, which is a multiple 8 (i.e., memory address `0x7ffdde8d0720`). Then, in a loop, starting from that address, your program prints out the byte pointed by the pointer, moves the pointer (++), prints out the byte, and moves the pointer again until it finishes all the arguments and reaches an 8B aligned address.

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. The command line below shows the parameters supported by the program. Note that the *f* option and the *s* option require option arguments. `Input_file` and `output_file` are operands.

```
./my_uniq -c -d -u -f fields -s char input_file output_file
```

Similar to the example program for `getopt`() in the slides, your program mainly calls `getopt`() repeatedly and interprets the results generated by `getopt`(), including the return values, `optopt`, `optarg`, and `optind` variables. Thus, it prints out the options, option parameters, and other arguments in the command line; if there are invalid options and/or option parameters missing, it prints out error messages.

Code beyond parsing the command line is not required. For example, your program does not need to determine whether the command line includes all the required options, or whether there are really files corresponding to `input_file` or `output_file`; it does not need to open or process the files.

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"
```

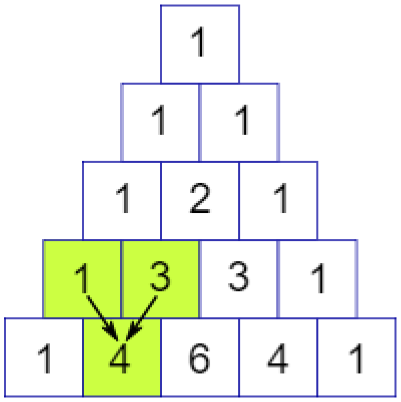
Your program needs to sort the environment variables based on their names. It can use *strtok()* to get the environment variable names. To sort the environment variables, it can exchange the pointers saved in the array pointed by `envp`. It can use bubble sort (https://en.wikipedia.org/wiki/Bubble_sort). So, it does not need to create other data structures. Strtok() replace '=' with NULL character. After it finishes sorting, it needs to put '=' back to where it was (strlen()).

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
```

Problem 4:

In C program, create and use a 2D dynamic array to compute, store, and print Pascal's triangle (refer to https://en.wikipedia.org/wiki/Pascal's_triangle). To build the triangle, start with generating the first row with 1 element and putting 1 into it. Then, generate and populate other rows of the 2D dynamic array, as shown below. The numbers on each row are calculated using the numbers on the previous row. For example, for the *i*-th row, its size is *i*; its first number is 1; the *j*-th number (for $1<j<i$) is the sum of $(j-1)$-th element and *j*-th element on the previous row, i.e., $(i-1)$-th row; the last number is 1.



When creating the 2D dynamic array, carefully control the size of each row to avoid wasting memory: on the first row, only allocate the space for 1 number, on the 2nd row, only allocate the space for 2 numbers, and so on. Failing to setup or use a 2D dynamic array (e.g., using a static array or doing calculation without using any array) will not receive any points.

The program takes the number of rows as its command line argument. You can assume that the number of rows does not exceed 100. So you can declare an array of pointers of size 100 in your program to organize the rows.

The following is the output of command `./your_program 6`

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```