Physical Participation Lab 1

Program Memory and Pointers, Debugging and Simulating Object Oriented Programming

Lab goals:

- C primer
- Understanding storage addresses, introduction to pointers
- Pointers to basic data types, to structures, and to functions
- Simulating object-like behavior in C

Remark : All the files mentioned in the lab text below are available on the course Moodle page.

(This lab is to be done SOLO)

Task 0: Using `gdb(1)` to debug segmentation fault

*You should finish this task **before** attending the lab session.*

C is a low-level language. Execution of a buggy C program may cause its abnormal termination due to *segmentation fault* --- illegal access to a memory address. Debugging segmentation faults can be a laborious task. GNU Debugger, is a powerful tool for program debugging and inspection. When a program is compiled for debugging and run inside gdb, the exact location of segmentation fault can be determined. In addition, the state of the processor registers and values of the variables at the time of the fault can be examined.

The source code for a buggy program, `count-words`, is provided in file count-words.c. (You can find the file in the course Moodle) The program works correctly most of the time, but when called with a single word on the command line, terminates due to segmentation fault.

1. Write a Makefile for the program.
2. **Specify compiler flags appropriate for debugging using** `gdb`.
3. Find the location and the cause of the segmentation fault using `gdb`.

4. Fix the bug and make sure the program works correctly.

The tasks below are to be done only during the lab session! Any code written before the lab will not be accepted.

Task 1: Understanding memory addresses and pointers

Logical virtual memory layout of a process is fixed in Linux. One can guess from the numerical value of a memory address whether the address points to:

- a static or a global variable,
- a local variable or a function argument,
- a function.

T1a - Addresses

Read, compile and run the addresses.c (You can find the file in the course Moodle) program (**remember to use the *-m32* flag**).
Can you tell the location (stack, code, etc.) of each memory address?
What can you say about the numerical values? Do they obey a particular order?

Check **long** data size on your machine using *sizeof* operator. Is *long integer* data type enough for **dist** (address difference) variables ?

T1b - Arrays memory layout

In this task we will examine the memory layout of arrays.
Define four arrays of length 3 as shown below *in the function main* and print the memory address of each array cell.

```
int iarray[3];
float farray[3];
double darray[3];
char carray[3];
```

Print the hexadecimal values of **iarray, iarray+1, farray, farray+1, darray, darray+1, carray and carray+1** (the values of these pointers, **not** the values pointed by the pointers). What can you say about the behavior of the '+' operator?

Given the results, explain to the TA the memory layout of arrays.

T1c - Distances

Understand and explain to the TA the purpose of the distances printed in the point_at function.
Where is each memory address allocated and what does it have to do with the printed distance? Given the results, explain to the TA the memory layout of arrays.

T1d - Pointers and arrays

Array names are essentially pointer constants. Instead of using the arrays, use the pointers below to access array cells.

```
int iarray2[] = {1,2,3};
char carray2[] = {'a','b','c'};
int* iarray2Ptr;
char* carray2Ptr;
```

Initialize the pointers iarrayPtr and carrayPtr to point to the first cell of the arrays iarray and carray respectively. Use the two pointers (iarrayPtr,carrayPtr) to print all the values of the two arrays.

Add an uninitialized pointer local variable p, and print its value (not the value it points to). What did you observe?

T1e - Address of command-line arguments

Add a prinout of the address and content of the command line arguments (argv, argv[0], argv[1], etc.), and run the program with some command-line arguments. What can you say about the memory location of the command-line arguments visible in main( )?

Task 2 - Structs and pointers to functions

Let us recall the following definition:

- **Pointers to functions** - C allows declaring pointers to functions. The syntax is: `function_return_type (*pointer_name)(arguments_list);` for simple types of return value and arguments. You can read more about pointers to functions here.

The base.c (You can find the file in the course Moodle) file is the base file for task 2 - you should complete it as stated in the sub tasks.

During Task 2 we read individual characters (bytes) from stdin using fgetc( )

- Please read the Deliverables section before continuing.

T2a

Implement the map function that receives a pointer to a char (a pointer to a char array), an integer, and a pointer to a function. Map returns a new array (after allocating space for it), such that each value in the new array is the result of applying the function f on the corresponding character in the input array.

1. char* map(char *array, int array_length, char (*f) (char))

Example:

```
char arr1[] = {'H','E','Y','!'};
char* arr2 = map(arr1, 4, xprt);
printf("%s\n", arr2);
free(arr2);
```

Results:

```
48

65

79

21
```

- Do not forget to free allocated memory.

T2b

Implement the following functions, and test them:

```
char my_get(char c);
/* Ignores c, reads and returns a character from stdin using fgetc. */

char cprt(char c);
/* If c is a number between 0x20 and 0x7E, cprt prints the character of ASCII
value c followed by a new line. Otherwise, cprt prints the dot ('.')
character. After printing, cprt returns the value of c unchanged. */

char encrypt(char c);
/* Gets a char c and returns its encrypted form by adding 1 to its value. If
c is not between 0x20 and 0x7E it is returned unchanged */

char decrypt(char c);
/* Gets a char c and returns its decrypted form by reducing 1 from its value.
If c is not between 0x20 and 0x7E it is returned unchanged */
```

```
char xprt(char c);
/* xprt prints the value of c in a hexadecimal representation followed by a
new line, and returns c unchanged. */
```

Note that array length is constant i.e. if the initial array is of length 5, then the new array that we receive with my_get function is of the same length.

Example:

```
int base_len = 5;
char arr1[base_len];
char* arr2 = map(arr1, base_len, my_get);
char* arr3 = map(arr2, base_len, cprt);
char* arr4 = map(arr3, base_len, xprt);
char* arr5 = map(arr4, base_len, encrypt);
free(arr2);
free(arr3);
free(arr4);
free(arr5);
```

Result:

```
Hey! // this is the user input.
H
e
y
!
.
48
65
79
21
.
```

- Do not forget to free allocated memory.
- There is no need to encrypt letters in a cyclic manner, simply add 1.

Task 3 - Menu

In this task we will be simulating objects in C. We will have a menu consisting of menu "objects", each of which has a name to be printed on the menu, and a "method" (which you will implement using a pointer to a function as a part of a "struct", as there are no "methods" or "objects" in C). **From here on we read complete lines from stdin (using fgets( ))instead of individual chars.**

struct - A struct in the C programming language is a structured type that aggregates a fixed set of labeled items, possibly of different types, into a single entity similar to an "object".

The struct size equals the sum of the sizes of its objects plus alignment (if needed). You can get the size by using the **sizeof** operator as follows: sizeof(struct struct_name).

T3a - Preliminary: Main Menu Loop and Terminating upon EOF.

Write and check a program called "menu" that prints to stdout a line stating "Select operation from the following menu:", reads an input line from stdin, and repeats forever unless it encounters an EOF condition for stdin. In the latter case, your program should exit normally.

Recall that typing 'Ctrl+D' in the console simulates an EOF condition for stdin.

T3b - Implementing the menu

In this task we will implement the menu. The menu must offer options for all your functions (Get String, Print String (crpt), Print Hex (xprt), Encrypt and Decrypt). See also the file Task 3 Example.

A function pointer can be a field in a structure, thus several functions can be accessed through a single data structure or container.

An array of function descriptors, each represented by a structure holding the function name (or description) and a pointer to the function, can be used to implement a program menu. Using the following structure definition:

```
struct fun_desc {
char *name;
char (*fun)(char);
};
```

Alternatively, you can define this as a "typedef".

Below is an example of declaration and initialization of a two-element array of "function descriptors":

```
struct fun_desc menu[] = { { "hello", hello }, { "bye", bye }, { NULL,
NULL } };
```

Extend your basic `menu` from T3a to offer the functions from task 2 in the following way:

1. Define a pointer 'carray' to a char array of length 5, initialized to the empty string (how?).

2. Defines an array of fun_desc and initializes it (in the declaration, not as program code within a function) to the names and the pointers of the functions that you implemented in Task 2. The last fun_desc in the array should contain a null pointer name and a null pointer to function (**the length of the array should not be kept explicitly after constructing it**).
3. Displays a menu (as a numbered list) of names (or descriptions) of the functions contained in the array. The menu should be printed by looping over the menu item names from the fun_desc, **not** by printing a string (or strings) that contain a copy of the name.
4. Displays a prompt asking the user to choose a function by its number in the menu, reads the number, and checks if it is within bounds. The bound should be pre-computed only **once**, and **before** the loop where the prompt is printed. If the number is within bounds, "Within bounds" is printed, otherwise "Not within bounds" is printed and the program exits gracefully.
5. Evaluate the appropriate function over 'carray' (using map) according to the number entered by the user. Note that you should call the function by using the function pointer in the array of structures, and not by using "if" or "switch".
6. After calling any menu function , let 'carray' point to the new array returned by map( ).
7. A user should be able to terminate the program when asked for a choice, by signaling EOF using 'Ctrl+D' on an empty line.

Task3 examples can be found in the course Moodle

Is it possible to call a function at an invalid address in your version of the program?

Bonus item (0 points) Add a menu item for "junk", where the pointer to function is initialized to point to something that is not known function code, such as your fun_desc array. Compile and run the modified program, and select the junk menu item. What do you observe?

Optional Tasks

Some topics in this course can be challenging, particularly for beginners , The "Optional Tasks" section of the lab is meant to encourage students to seek assistance from the TAs and to ensure you comprehend the parts do-at-home labs (in this case Lab A and Lab B) with which we believe you may face

difficulties. If you and the TA have free time during the lab, you may wish to do the following tasks from these labs during lab 1.

Makefile task(Optional 1, from Lab A)

Correct detection of EOF (Optional 2, from Lab A)

Using Valgrind (Optional 3, from Lab B)

Deliverables

As for all labs, you should complete task 0 before the lab, and make sure you understand what you did.

During the lab, you should complete all tasks, 1, 2, and 3. If you did not complete task 3 submit whatever you did complete in the lab. Your lab grade is determined and decided in most cases by your lab TA during the lab. Still, you are required to submit your code at the end of the lab, and the grade may be adjusted based on what you submit. For example, failure to quote code copied from elsewhere, even a permitted source, could cause a reduced grade, or worse.

The deliverables must be submitted until the end of the day.

You must submit source files for task3 (note that task3 includes the functions in task 2) in respective folders, and also a makefile that compiles them. The source files must be organized in the following tree structure (where '+' represents a folder and '-' represents a file):

+ task3
  - makefile
  - menu_map.c

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.