C Programming: Program Memory, Debugging Fatal Errors, Function Pointers

(This lab to be done SOLO)

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

*You should accomplish 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.

`gdb(1)`, the 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. 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 compilation 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.

Tasks below to be done in lab sessions only!

Task 1: Analyzing memory addresses

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

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

Here is a useful link (in addition to what you've heard in class)

T1a - Addresses

Read, compile and run the addresses.c program.
Could you tell the location of each memory address?
What can you say about the numerical values? Do they obey a particular order?

T1b - Distances

Understand and explain the TA the purpose of the distances printed in the point_at function.
When is each memory address allocated and what does it have to do with the printed distance?

T1c - Deeper understanding of addresses (bonus task - 5 points)

In C, **without** using "union", declare two **different** variables (say x and y) that have the same address. Prove that they have the same address by printing it.

T2 - Filter function

T2a

Write two functions that receive a "char*" argument and return an "int", with the following specification:

1. First function: returns the number of chars in the char*
2. Second function: returns the number of spaces in the char*

NOTE: a proper string ends with a special character, ('\0').These strings are called null terminated strings.

T2b

Write a filter function myFilter with the following specification:
Receives 3 arguments: an array of pointers (to char*), the size of the array, and a pointer "f" to a function that receives a char* argument and returns int.
The pointer to a function of this type is declared as follows: `int (*func)(char*);`
Description: myFilter applies the function pointed to by "f" to each of the elements of the array, and prints the elements for which the function returns a number greater than 4.

Test your code by reading arrays of "character strings" from the user, and using at least one of the functions from T2a as "arguments" to myFilter.

T2c

A function pointer can also be a field of a structure, and thus several functions can be held in a single data structure, or container (This is one mechanism for implementing methods of objects in the internals of an object-oriented paradigm).

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:

```
1. struct fun_desc {
2.   char *name;
3.   void (*fun)(void);
4. };
```

Alternately, you can define this as a "typedef" as shown in class.

Write a modified version of the program from task 2b that:

1. Defines an array of fun_desc and initializes it to names and pointers of the two functions implemented by you in T2a. (check that your struct is using the right types!)
2. Displays a menu (as a numbered list) of names (or descriptions) of the functions contained in the array.
3. 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. (If not, either prompt the user again, or exit gracefully).
4. Displays a prompt asking the user to insert an argument for the selected function.
5. Calls the appropriate function 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".

Example for bullet 3:

```
Please choose a function:
0) Characters counter.
1) Spaces counter.
```

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

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

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

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 at least task 1 and task 2a,b and as much as possible from task 2c. If you cannot finish task 2c before the end of the lab, you should complete it during a make-up lab. There is no penalty for not completing task 2c during the first lab session, provided you came prepared, on time, and worked seriously on the tasks for the entire duration of the lab.