

Program Memory and Pointers, Debugging and Simulating Object Oriented Programming

Lab goals:

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

(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.

`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.

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](#) 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 iarray[] = {1,2,3};
char carray[] = {'a','b','c'};
int* iarrayPtr;
char* carrayPtr;
```

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?

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` file is the base file for task 2 - you should complete it as stated in the sub tasks.

- Please read the Deliverables section before continuing.

Task 2a

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, censor);
```

```
printf("%s\n", arr2);
free(arr2);
```

Results:

HEY.

- Do not forget to free allocated memory.

Task 2b

Implement the following functions.

```
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 3 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 3 to 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. */
```

Pay attention 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);
char* arr6 = map(arr5, base_len, decrypt);
free(arr2);
free(arr3);
free(arr4);
free(arr5);
free(arr6);
```

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/decrypt letters in a cyclic manner, simply add/reduce 3.

T2c - Adding an option to exit

Implement the following function:

```
char quit(char c);  
/* Gets a char c, and if the char is 'q' , ends the program with exit  
code 0. Otherwise returns c. */
```

This function terminates the program "normally" and "successfully" using the *exit* function (as mentioned in the lab's reading material). The use of such a function will be clarified in task 3.

Task 3 - Menu

- **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).

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 } };
```

Using the code from 2c, write a program called `menu` that performs the following.

1. Defines 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 (other than 'quit'), let 'carray' point to the new array returned by `map()`.

Press here for examples - [Task3-Examples](#)

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?

the quit function

In task 2c we have defined the quit function as a function that gets and returns a char. This is an unusual implementation, however, it enabled us to nicely add a quit option to the menu that follows the same architecture of the assignment. We did not have to explicitly write a separate menu item for the quit option. Be that as it may, notice that it is a quick and dirty "trick" and it is **not** the generally recommended way of constructing menus.

- Instead of performing the actions one at a time, save all the chosen actions in an array (of size 10), then when the user chooses to quit, carry out all the actions and quit. (To be done only if you need a "special" alteration.)

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 2 and as much as possible from task 3. If you cannot finish task 3 before the end of the lab, you should complete it during a make-up lab.

There is no penalty for not completing task 3 during the first lab session, provided you came prepared, on time, and worked seriously on the tasks for the entire duration of the lab.

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

You must submit source files for task2C and task3 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):

```
+ task2C
  - makefile
  - menu_map.c
+ 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.