Program Memory and Pointers, Debugging and Simulating Object Oriented Programming

Lab goals: C primer, pointers to data, structures, and pointers to functions.

(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.

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

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?

T1b - 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?

T1c - Arrays memory layout

In this task we will examine the memory layout of arrays.

Define two arrays of length 3 as shown below and print the memory address of each array cell.

```
int iarray[3];
char carray[3];
```

Print the hexadecimal values of **iarray, iarray+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.

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. 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);`. You can read more about pointers to functions here .

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

```
                                        14 lines ...

#include <stdio.h>

int plus_one(int n) {
  return n+1;
}

int* map(int *array, int arrayLength, int (*f) (int)){
  int* mappedArray = (int*)(malloc(arrayLength*sizeof(int)));
  /* TODO: Complete during task 2.a */
  return mappedArray;
}

int main(int argc, char **argv){
  /* TODO: Test your code */
}
```

Task 2a

Implement the map function which receives a pointer to an int (a pointer to an int array), the number of elements in the array, 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 number in the input array.

```
1. int* map(int *array, int arrayLength, int (*f) (int));
```

Example:

```
int len = 4;
int i;
int arr1[] = {5, -2, 7, 8};
```

```
int* arr2 = map(arr1, len, plus_one);
for(i=0 ; i<len ; i++)
   printf("%d,\n", arr2[i]); /* 6, -1, 8, 9, */
free(arr2);
```

- Do not forget to free allocated memory.

Task 2b

Implement the following functions.

```
int abs(int n); /* Gets an integer n, and returns the absolute value of n. */
int iprt(int n); /* Prints the value of n followed by a new line, and returns n unchanged */
int cprt(int n); /* Prints the character of ASCII value n followed by a new line, and returns n unchanged.
                    If n is not between 0x20 and 0x7E, print the dot ('.') character instead. */
int my_get(int n); /* Ignores n, reads a line from stdin, and returns a number given in that line.  */
```

Example:

```
                                        11 lines ...

int len = 4;
int arr1[len];
int* arr2 = map(arr1, len, my_get);
int* arr4 = map(arr3, len, abs);
int* arr3 = map(arr2, len, iprt);
int* arr5 = map(arr4, len, plus_one);
int* arr6 = map(arr5, len, cprt);
free(arr2);
free(arr3);
free(arr4);
free(arr5);
free(arr6);
```

```
Result:
2
-40
1
-55
2
40
1
55
.
)
.
8
```

- Do not forget to free allocated memory.

T2c - Adding an option to exit

Implement the following function:

```
int quit(int n); /* Gets an integer n, and ends the program using n as the return value */
```

This function ends the program using the *exit* system call (as mentioned in the lab's reading material). The use of such a function will be clarified in task 3.

- There is no need to do anything other than call the exit function

Task 3 - Menu

- **struct -** A struct in C programming language is a structured type that aggregates a fixed set of labelled objects, possibly of different types, into a single "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 held in 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;
  int (*fun)(int);
};
```

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

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

1. Defines an int array named 'iarray' of length 4.
2. Defines an array of fun_desc and initializes it 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 'iarray' (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".

Usage Example:

```
#> menu
Please choose a function:
0) Plus One
1) Abs
2) Print Integer
3) Print Character
4) Get numbers
5) Quit
Option: 4
within bounds
2
-40
1
-55
DONE.

Please choose a function:
0) Plus One
1) Abs
2) Print Integer
3) Print Character
4) Get numbers
5) Quit
Option: 1
```

```
within bounds
DONE.

Please choose a function:
0) Plus One
1) Abs
2) Print Integer
3) Print Character
4) Get numbers
5) Quit
Option: 2
within bounds
2
40
1
55
DONE.

Please choose a function:
0) Plus One
1) Abs
2) Print Integer
3) Print Character
4) Get numbers
5) Quit
Option: 0
within bounds
DONE.

Please choose a function:
0) Plus One
1) Abs
2) Print Integer
3) Print Character
4) Get numbers
5) Quit
Option: 2
within bounds
3
41
2
56
DONE.

Please choose a function:
0) Plus One
1) Abs
2) Print Integer
3) Print Character
4) Get numbers
5) Quit
Option: 3
within bounds
.
)
.
8
DONE.

Please choose a function:
0) Plus One
1) Abs
2) Print Integer
3) Print Character
4) Get numbers
5) Quit
```

```
Option: 5
within bounds
```

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 } };
```

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

> **The quit function**
> In task 2c we have defined the quit function as a function that gets and returns an int. 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.

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?

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 and appropriate makefiles for tasks 2c and 3. The source files and their respective makefiles must be named **task2c.c**, **makefile2c**, **task3.c** and **makefile3**.

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.