# Program Memory and Function Pointers. Simulating Object Oriented Programming.

# (This lab is to be done SOLO)

# Overview

In this lab you will edit the state machine from lab 2. The goal of the lab is to make that state machine more modular using function dispatching. In C, which is not an object oriented programming language (that is, no classes, objects, inheritance, etc.), this can be accomplished through Function Pointers.

At the end of this lab (after Task 2), your state machine will turn into the following simple one:

```
- get the current state.
- retrieve the appropriate handler of the current state.
- call the handler of the current state.
```

## Task 0: Replacing the statements in the switch case in the state machine

- In this task, you are required to implement a handler for each state in your state machine (e.g., init, percent, and any other state you defined in your implementation). All of the functions should be of the following prototype:

```
typedef struct {
    char* fs;
    /* Any extra required args */
} state_args;

enum printf_state function_name(va_list args, int* out_printed_chars, state_args* state);
```

The function takes the variable arguments list from `toy_printf`, an out int parameter to output the number of chars printed by the handler and extra arguments required for the state to be read or updated by the handler.

You have to replace the body in each case of your state machine to a call to the relevant handler. Your `toy_printf` should look a bit like this:

```
int toy_printf(fs, ...) {
  .
  .
  .
  switch (state) {
    case (st_printf_init):
      state = init_state_handler(args, &handler_printed_chars, state_args);
      printed_chars += handler_printed_chars;
      break;
    case(st_printf_percent):
      state = percent_state_handler(args, &handler_printed_chars, state_args);
      printed_chars += handler_printed_chars;
      break;
    .
    . // Cases for any extra states defined in printf_state
```

```
    .
  }
  return printed_chars;
}
```

> **Note that all the logic (printing, reading args, updating state_args and choosing the next state) should be handled by the state handlers, not** `toy_printf`. Code which includes state-specific logic in `toy_printf` will not be accepted as well as will be harder to work with in later tasks.

## Task 1: Understanding memory addresses and pointers

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.
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 meaning 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[] = {0x01234567,0x89ABCDEF,0x13579BDF};
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 - Dispatched state machine

## Task 2a

The functions you wrote in Task 0 have two output values. The first is the next state, which is returned through the return statement. The second is the number of printed characters, which is returned through the pointer parameter. In this task, you will refactor the return type of these handlers to return a single value, a struct type.

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

Use the following structure definition to replace the return values of all the functions, and change the statements in the switch case accordingly. The following struct represents the result of a single state execution.

```
struct state_result {
  int printed_chars;
  enum printf_state new_state;
};
```

- **Function Pointers** - 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 .

## Task 2b

In this task you are required to replace the "switch-case" in the State Machine by one call to the appropriate handler of each state. To accomplish this, you will use an array of function pointers.

- Define an array of size as the number of the states in the state machine. Each cell in the array holds a handler. Each enum value is is mapped to a numerical value, so each index in the array will describe the handler for that state. e.g. `array[st_printf_init]` will hold the handler for init state
- Replace the "switch-case" in the state machine by calling the appropriate function which you retrieve from the array. **Note, you should not have if-else chains or switch-case in the state machine anymore (you can use switch-case in the handlers).**

## Task 2c

Now, you are required to replace the switch-case in the handlers implementation in a similar way. For instance, in the previous task you probably had the following code in the st_printf_percent case:

```
state_result percent_handler(va_list args,int* out_printed_chars, state_args* state) {
  //...
  switch (*fs) {
    case 'd' :
      //some work
      break;
    case 's':
      //...
  }
}
```

Now, you are required to remove this switch case from the the handlers. For the st_printf_percent You have to define a handler for each identifier (e.g, d, o, s, x, etc). Like in task 3a, and create a handlers array. Follow these steps:

- Write a handler for each identifier.
- Define an array of size 128 cells (a cell for each ascii char). Each cell in the array holds either a pointer to a handler of an identifier or a pointer to a default handler (for unknown identifiers). So, for instance, `array['d']` will hold the handle for printing a signed decimal int.

- Replace the "switch-case" in the handlers state by calling the appropriate identifier handler which you retrieve from the array.

  If you have any handlers using switch case or if-else chains, change them accordingly to use function dispatching. **Note, you should not have if-else chains or switch-case in handlers anymore.**

## Deliverables:

Task 1[a-d] and 2[a-b] must be completed during the regular lab. Task 2c may be done in a completion lab if you run out of time during the regular lab. The deliverables must be submitted until the end of the lab.
You must submit source files 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):

+ task2b
  - makefile
  - toy_printf.c
  - main.c
  - toy_stdio.h
+ task2c
  - makefile
  - toy_printf.c
  - main.c
  - toy_stdio.h

Note that the folder name "task2b" should be used, regardless of whether you completed task2b or not. You only need to include folder "task2c" if you completed task2c.

### 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.