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.
5. By the end of this task you should be familiar with the following GDB features : Breakpoints, Watchpoints, Source listing, Stepping, and Printing variables and memory

The tasks below are to be done only during the lab session!

Task 1: Extend toy_printf function

The function printf() and its cousins are among of the most versatile and well-known functions for sending formatted strings as output to files and the screen. They are implemented by many languages, including C, Java, Perl, PHP, Python, Ruby, and a handful of others.

One of printf()'s signature benefits is its ability to accept a variable number of arguments. This allows printed text to contain countless variations of formatted values and strings. In this task, you will learn how to extend this functionality to create your own printf() wrapper function. The source code you need to modify in this lab is provided in the files toy_printf.c, main[1].c, toy_stdio.h

toy_printf's manual

```
Synopsis
#include "toy_stdio.h"

int toy_printf(char* format, ...);

Description
toy printf outputs to stdout according to the given null-terminated format string
(see below) and returns the number of characters written.
The format string may contain format flags and which correspond to additional arguments
passed to toy_printf. The position of the format flag corresponds to the position
(after format) of its related argument.
```

```
Format flags
A format flag is a percent ('%') sign followed by one of the following format identifiers.
In the output, the format flag is replaced with string representing its related argument.

identifiers:
d       signed integer as a decimal number
u       unsigned integer as a decimal number
b       unsigned integer as a binary number
o       unsigned integer as a octal number
x       unsigned integer as a hexadecimal number. Letter numerals printed in lower case
X       unsigned integer as a hexadecimal number. Letter numerals printed in upper case
s       null-terminated string
c       character according to the ascii table
%       the '%' character.
```

Task 1a:Print unsinged

Fix the output for integers of numbers in bases other than base 10, so that the numbers are printed as unsigned rather than signed. For example:

```
toy_printf("Hex unsigned: %x\n", -1);
toy_printf("Octal unsigned: %o\n", -1);
```

Output:

```
#> main
Hex unsigned: ffffffff
Octal unsigned: 37777777777
#>
```

T1b- Print unsigned integers

Add support for printing unsigned integers %u ("int" argument is output as an unsigned decimal integer). For example:

```
toy_printf("Unsigned value: %u\n", 15);
toy_printf("Unsigned value: %u\n", -15);
```

Output:

```
#> main
Unsigned value: 15
Unsigned value: 4294967281
#>
```

T1c - Printing arrays

Add support for %Ad, %As, %Ax, etc., to print arrays of integers, string, hexadecimal integers, etc. The size of the array is passed as an argument. For example:

```
int integers_array[] = {1,2,3,4,5};
char * strings_array[] = {"This", "is", "array", "of", "strings"};
int array_size = 5;
toy_printf("Print array of integers: %Ad\n", integers_array, array_size);
toy_printf("Print array of strings: %As\n", strings_array, array_size);
```

Output:

```
#> main
Print array of integers: {1, 2, 3, 4, 5}
Print array of strings:  {This,  is,  array,  of,  strings}
#>
```

T1d - The Width Field

The width field is a non-negative decimal integer giving the minimum number of characters to be printed. If the output value is shorter than the given width, it is padded to the appropriate width by putting blanks on the right (or on the left, if the '-' "modifier" character is specified). Add support for width of fields %<width>d, %<width>s. Please note, when padding blanks on the right you should end the padding with an '#' character.

For example:

```
toy_printf("Non-padded string: %s\n", "str");
toy_printf("Right-padded string: %6s\n", "str");
toy_printf("Left-added string: %-6s\n", "str");
```

Output:

```
#> main
Non-padded string: str
Right-padded string: str   #
Left-padded string:    str
#>
```

With numeric placeholders, the number in the width field may have a leading 0. With this, the output value will be expanded with zeros to give the number the specified width. Add support for numeric fields filled with initial zeros: %0<width>d. For example, with "%05d" the value -1 will be printed as "-0001".

```
toy_printf("With numeric placeholders: %05d\n", -1);
```

Output:

```
#> main
With numeric placeholders: -0001
#>
```

Deliverables:

Task 1[a-c] must be completed during the regular lab. Task 1d may be done in a completion lab, but only 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):
+ task1d
  - makefile
  - toy_printf.c
  - main.c
  - toy_stdio.h

Note that the folder name "task1d" should be used, regardless of whether you completed task1d or not.

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.