B1 - Unix System Programming

B-PSU-100

my_printf

Bootstrap

2.1

my_printf

binary name: libmy.a
language: C
compilation: via Makefile, including re, clean and fclean rules

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.

During this bootstrap we will come back on the notion of **static library** creation and we are going to explore how to use **va_args** from the **stdarg.h** headers.

Step 1: Static library.

First we will setup your folder architecture, which must look like that:

```
> tree
.
|-- includes
|    `-- bsprintf.h
|-- Makefile
|-- sources
|    |-- disp_stdarg.c
|    |-- sum_numbers.c
|    `-- sum_strings_length.c
`-- tests
     |-- tests_disp_stdarg.c
     |-- tests_sum_numbers.c
     `-- tests_sum_strings_length.c
```

You will implement functions with the following prototypes in their corresponding files:

```
int sum_numbers(int n, ...);
int sum_strings_length(int n, ...);
void disp_stdarg(char *s, ...);
```

You should add those prototypes to the `includes/bsprintf.h` file (do not forget about the include guard in your header).
In your `Makefile` you must list all of the `C` files separatly (no `*.c`), compile them, and use the result to generate the `libmy.a` library.
Your `Makefile` must have the following rules:

- libmy.a
- all (which calls the libmy.a rule)
- clean
- fclean (which calls the clean rule)
- re (which calls the fclean and libmy.a rules)
- unit_tests (which calls the fclean and libmy.a rules, and then links the lib with the tests)

- run_tests (which calls the unit_tests rule and executes the unit_tests bin)

Don't hesitate to write the tests before adding any features to your program. TDD

STEP 2: VA_ARGS

Before attempting the next exercices, please take a look at the following man pages and this video.

```
> man va_arg
> man stdarg.h
```

PART A: SUM NUMBERS

```c
/*
** The sum_numbers() function returns the sum of the numbers passed as parameters
   after n.
** The parameter 'n' represents the number of arguments that will be passed as
   parameter.
** During our tests, the parameter 'n' value will always be lesser or equal to the
   number of parameters given (never greater).
*/
int sum_numbers(int n, ...);
```

PART B: SUM STRING LENGTH

```c
/*
** The sum_strings_length() function returns the sum of the lengths of every string
   passed as parameter after n.
** The parameter 'n' represents the number of arguments that will be passed as
   parameter.
** During our tests, the parameter 'n' value will always be lesser or equal to the
   number of parameters given (never greater).
*/
int sum_strings_length(int n, ...);
```

PART C: DISPLAY STDARGS

```c
/*
** This function displays all of its arguments (except the first one),
**  each followed by a '\n', in the order in which they were passed.
** The value of each char composing the first parameter given tells you if the next
** arguement is a char (if the next character is a 'c'), a char* ('s') or an int (an
   'i').
*/
void disp_stdarg(char *s, ...);
```

Only malloc, free and write are allowed.

Unit Tests

It's generally considered a good practice to write unit tests before starting to implement a function. Think about all the cases you should be able to handle!

Here are some basic tests to dispatch to each test files.

```c
#include <criterion/criterion.h>
#include <criterion/redirect.h>

Test(sum_numbers, return_correct_when_i_is_zero)
{
    int ret = sum_numbers(3, 21, 25, -4);
    cr_assert_eq(ret, 42);
}

Test(sum_numbers, sum_integer_values) {
    int value = sum_numbers(3, 1, 2, 3);
    cr_assert_eq(value, 6);
}

Test(sum_strings_length, sum_str_lengths) {
    int value = sum_strings_length(5, "Hello", "a", "toto", "", "plop");
    cr_assert_eq(value, 14);
}

Test(disp_stdarg, basic, .init=cr_redirect_stdout) {
    disp_stdarg("csiis", 'X', "hi", 10, -3, "plop");
    cr_assert_stdout_eq_str("Xnhin10n-3nplopn", "");
}
```