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 how to create a **static library** than we are giong to explore how to use **va_args** from the **stdarg.h** headers.

Step 1: Static library.

First we will setup your folder tree, it must look like that:

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

Implement the following prototypes to the corresponding file:

```
int sum_numbers(int n, ...);
int sum_strings_length(int n, ...);
void display_stdarg(int n, ...);
```

You must also add the previous prototypes to the `includes/bsprintf.h` file (do not forget about the include guard in your header).
Then in your `Makefile` you must do compile each of the `C` files, each of them must be compiled separatly (no `*.c`) and generate the `libbsprintf.a` library.
Your `Makefile` must have the following rules:

- libbsprintf.a
- all (which call the rule libbsprintf.a)
- clean
- fclean (which call the rule clean)
- re (which call the rule fclean and then libbsprintf.a)
- unit_tests (which call the rule fclean and then libbsprintf.a, then link the lib with the tests)

- run_tests (which call the rule libbsprintf.a and execute the unit_tests bin)

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

STEP 2: VA_ARGS

Please before doing any parts take a look to the following man pages and this video.

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

PART A: SUM NUMBERS

```
/*
** The sum_numbers() function return the sum of the numbers passed as parameters.
** the parameter 'n' represent the number va_args passed as parameters.
** during our tests, the parameter 'n' value will always be at most the number of
    parameter (never more).
*/
int sum_numbers(int n, ...);
```

PART B: SUM STRING LENGTH

```
/*
** The sum_strings_length() function return the sum of each string passed as
    parameters.
** the parameter 'n' represent the number va_args passed as parameters.
** during our tests, the parameter 'n' value will always be at most the number of
    parameter (never more).
*/
int sum_strings_length(int n, ...);
```

PART C: DISPLAY STDARGS

```
/*
** This function displays the arguments followed by '\n',
** in the order in which they were passed, according to the value of s,
** which is composed of the letters c (for char), s (for char*) and i (for int).
*/
void disp_stdarg( char *s, ... );
```

Only malloc, free and write are allowed.

Unit Tests

It's generaly considered good practice to write its unit tests before starting to implement a function. Think of all the cases!

Here' some basic tests to dispatch between each tests 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", "");
}
```