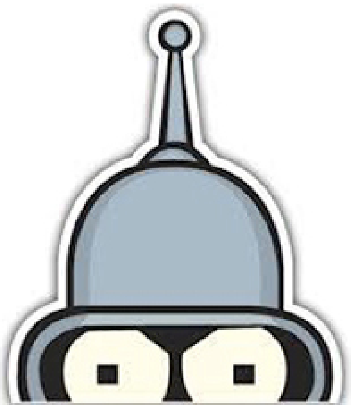

B1- Elementary Programming in C

B-CPE-110

Bootstrap

BSQ

1.8

Bootstrap

BSQ

repository name: : CPE_BSQ_bootstrap_$ACADEMICYEAR
repository rights: : ramassage-tek
language: : C
group size: : 1

Each exercise is to be submitted in a C file with the same name as the function in the repository's root. If one of your files prevents you from compiling with *.c, the Autograder will not be able to correct your work.
Don't push your main function into your delivery directory, we will be adding our own. Your files will be compiled adding our main.c.

The BSQ project consists in recovering a map with obstacles and finding, on this map, the largest possible square. That's why this bootstrap covers a lot of different things: files, 2-D arrays, string searches,...

For the first part of this bootstrap, you will be working on reading files.
Man pages open(2), read(2), write(2), and close(2) are your friends.

After that, we will turn you into 2d-array pathway and allocation rock stars. Man pages malloc(3) and free(3) will help you.

We will then move on to a few algorithms. This part will focus on creating the necessary tools in order to complete a basic version of the BSQ.

And finally, for those who are highly-motivated, we will conclude the BSQ by introducing graphical restrictions.

I- File systems

Step 1

Write a function that opens a read-only file, which displays "SUCCESS\n" or "FAILURE\n" (depending on the return value), and which returns the file descriptor.
The function should be prototyped the following way:

```
int fs_open_file(char const *filepath)
```

Take some time to understand what a file descriptor is, and the difference between the different flags you can come across in open, as second parameter.

Step 2

The next function must display a specific message according to the read return.
The function should be prototyped the following way:

```
void fs_understand_return_of_read(int fd, char *buffer, int size)
```

The parameters are the same parameters as those passed to read.
Here are the messages to display:

- if the return value is -1, display "read failed\n",
- if the return value is 0, display "read has nothing more to read\n",
- if the return value is smaller than size, display "read didn't complete the entire buffer\n",
- if the return value is equal to size, display "read completed the entire buffer\n".

Step 3

Now that you know how to open a file, what a file descriptor is and what the different return values possible for read mean, we're going to put it all together.
You are to write a function that displays the first 500 bytes of a file passed as a parameter.
The function should be prototyped the following way:

```
void fs_cat_500_bytes(char const *filepath)
```

! Don't forget to close the opened file!

Step 4

Write now a function that displays the first x octets of a file passed as the first parameter, where x is the second parameter.
The function should be prototyped the following way:

```
void fs_cat_x_bytes(char const *filepath, int x)
```

Step 5

When you read a file, sometimes more information needs to be extracted. This can be achieved through **data clusters** (see previous steps) or by applying a different rule.
For example, you can decide that each piece of information will be written on one of the file's lines.
Write a function that displays the first line of a file passed as parameter (without the $\backslash n$).
The function should be prototyped the following way:

```
void fs_print_first_line(char const *filepath)
```

Step 6

You are going to combine everything you've seen so far in order to write a function that will retrieve and return the positive number written on the first line of the file passed as parameter.
If an error occurs (file to be read inexistant or inaccessible, content of the first line that is not a number, a number found that isn't strictly positive,..), return the **-1** value. The function should be prototyped the following way:

```
int fs_get_number_from_first_line(char const *filepath)
```

II- 2d-arrays

Step 1

First, let's display the array's values line by line, with one character per line.
The function should be prototyped the following way:

```
void array_1d_print_chars(char const *arr)
```

Step 2

Let's display once more the array's values line by line, one character per line, but from an int array (not from a char array anymore).
If you are comfortable with **my_put_nbr**, nothing complicated.

Food for thought: in a string, you wrap until you find "\0", which indicates the end. But when you have to scan an array without an end indicator, when should you stop?

The function should be prototyped the following way:

```
void array_1d_print_ints(int const *arr, int size)
```

Step 3

Write a function that adds the values contained in the 1d-array and returns the result.
The function should be prototyped the following way:

```
int array_1d_sum(int const *arr, int size)
```

Step 4

Now write a function that adds the values contained in the 2d-array and returns the result.
The function should be prototyped the following way:

```
int array_2d_sum(int **arr, int nb_rows, int nb_cols)
```

Step 5

Count the number of times **NUMBER** appears in the array and return the result.
The function should be prototyped the following way:

```
int array_2d_how_many(int **arr, int nb_rows, int nb_cols, int number)
```

III- Memory Allocation

Step 1

Write a function that allocates a memory zone with a size capable of containing strings **a** and **b** (passed in parameters), and then copy the linked strings in this new memory zone.
Your function will return the new zone's address and should be prototyped the following way:

```
char *mem_alloc(char const *a, char const *b)
```

Take some time to understand the benefit of the **sizeof** keyword.

Step 2

Now write a function that allocates a 2d-array and returns its address.
The function should be prototyped the following way:

```
char **mem_alloc_2d_array(int nb_rows, int nb_cols)
```

Step 3

Write a function that duplicates a 2d-array.
The function should be prototyped the following way:

```
char **mem_dup_2d_array(char **arr, int nb_rows, int nb_cols)
```

IV- Algorithms

Overview

Let's get down to business. We manipulated files, saw how to allocate memory for 1d and 2d-arrays, and also practiced scanning. It's time to put it all together and see how to create nice tools that will be useful for our BSQ.

Step 1

At some point the BSQ project will require to load a map in memory.
Write a function that takes a file as parameter, load its content in memory and returns the address of this memory area.
The function should be prototyped the following way:

```
char *load_file_in_mem(char const *filepath)
```

Step 2

For our purposes, the file passed as parameter now contains a rectangle. Therefore, we want to load this rectangle into a 2d-array.
The function should be prototyped the following way:

```
char **load_2d_arr_from_file(char const *filepath, int nb_rows, int nb_cols)
```

Step 3

Now that the map can be loaded in memory as a char**, let's manipulate it. Write a function that returns 1 if a square of a given size can be found in the position indicated as parameter (**row, col**), and 0 otherwise. The function should be prototyped the following way:

```
int is_square_of_size(char **map, int row, int col, int square_size)
```

You should work under the assumption that the map here is similar to the one introduced in the BSQ project. i.e. containing '.' and 'o'. A square should not contain any 'o'.

Step 4

Write a function that finds the biggest possible square in a map, its upper lefthand corner being positioned as indicated as parameter (**row, col**).
Your function should handle any errors that appear and be prototyped the following way:

```
int find_biggest_square(char **map, int nb_rows, int nb_cols, int row, int col)
```

Almost the end...

Since you've made it this far, you are now a master of BSQ concepts! Here are a couple of tips to ace this project:

- what is the smartest data structure to store the lines read from the file?
- what is the fastest way to look for big square in a map?
- how can you measure the time spent to complete the task? Are there any useful UNIX command?
- what if the file you're given is incorrect? How to check thoroughly all the irregular cases?
- did you close and free carefully?

For those who are highly motivated, we've saved you one more exercise. It's not directly related to BSQ, but it can give you some good bonus ideas!

Going further away

Write a function that takes a rectangular map, along with its size, and a 2d-array (**shape**) with its size. In this 2d-array, a shape is drawn (a space ' ' representing an empty cell, and all other characters except '\0' representing a full cell).
The function returns 1 if the shape can be found somewhere in the map without any obstacles, and 0 otherwise.

```
int can_draw_shape_in_map(char **map, int map_nb_rows, int map_nb_cols, char **shape, int
    shape_nb_rows, int shape_nb_cols)
```

There are plenty of ASCII art drawings online that could be used as examples. Help yourself :)