C1- Unix & C Lab Seminar

pool_c_d09

Day 09-10

Rubik's Square

CODING ACADEMY
BY EPITECH

Instructions

Before You Go Further

- Turn in directory **pool_c_d09**.
- Respecting the norm takes time, but it is good for you. This way your code will respect the norm from the first written line to the last.
  Read carefully the norm documentation. You should type “alt+i” instead of “tab”
- You shall leave in your directory no other files than those explicitly specified by the exercices. If one of your files prevents the compilation with *.c, the robot will not be able to do the correction and you will have a 0.
- **Do not to turn-in a main() function.**

Introduction

Today, you will have to play with a square on the same principle as the rubik's cube but on a two dimensional square.
We want you to understand what you do, everything today is about algorythm, it's not a rought word, don't worry.
The puzzle is split step by step, there are many ways to do it, this solution is certainly not the best but it might be the most pedagogical.
After each exercise is done, turn it in and then, copy the folder for the next exercise. And, keep you header file (rubiks.h) up to date.
This is not an easy exercice, but if you follow the exercice and ask the geckos for help, you might succeed.

Exercise 1

Print And Check (1 pt)

Turn in: pool_c_d09/ex_01/ruler.c
pool_c_d09/ex_01/rubiks.h

Prototype: void print_tab(int **table);
Prototype: int check_square(int **table);

Write a function 'print_tab' taking a pointer of pointer on integer as parameter and returning nothing. This parameter is a four columns by four rows table. You have to allocate it (with "malloc()") in your main().
This function will produce the following output :

```
| 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 |
| 2 | 2 | 3 | 3 |
| 2 | 2 | 3 | 3 |
```

Take care, the cosmetic around the square is compose with dash and pipes

Exercise 2

Algorithms (2 pts)

Turn in: pool_c_d09/ex_02/algo.c

Prototype: void algo_line(int **table, int line);
Prototype: void algo_column(int **table, int column);
Prototype: void algo_square(int **table, int square);

Write three algorithm to play with your magic square :

- algo_line, taking the table and the line number as parameter.
  This function must rotate this single line on the **left**.
- algo_column, taking the table and the column number as parameter.
  This function must rotate this single column on the **Top**.
- algo_square, taking the table and the square number as parameter.
  This function must rotate this single square **Clockwise**.

You must also create a define PRINT_SQUARE_DEBUG__.Do not let it in your code, we will add it in the correction.
If this define is set to 1, you have to display the rotation way and the square, otherwise, you should not.

PRINT_SQUARE_DEBUG__ will always be defined. You have to check its value to decide if you have to display or not

Example :

```c
#define PRINT_SQUARE_DEBUG__ 1
// ...
int     main()
{
  int   **table;
  // ...
  algo_line(table, 0);
  algo_column(table, 0);
  algo_square(table, 0);
  // ...
}
// ...
```

This piece of code will produce the following output :

```
Rotate Left line 0.
_____________
| 0 | 1 | 1 | 0 |
_____________
| 0 | 0 | 1 | 1 |
_____________
| 2 | 2 | 3 | 3 |
_____________
| 2 | 2 | 3 | 3 |
_____________

Rotate Top column 0.
_____________
| 0 | 1 | 1 | 0 |
_____________
| 2 | 0 | 1 | 1 |
_____________
| 2 | 2 | 3 | 3 |
_____________
| 0 | 2 | 3 | 3 |
_____________

Rotate Clockwise square 0.
_____________
| 2 | 0 | 1 | 0 |
_____________
| 0 | 1 | 1 | 1 |
_____________
| 2 | 2 | 3 | 3 |
_____________
| 0 | 2 | 3 | 3 |
_____________
```

Exercise 3

Reverse Algorithm (2 pts)

Turn in: pool_c_d09/ex_03/algo_rev.c
Prototype: void algo_line_reverse(int **table, int line);
Prototype: void algo_column_reverse(int **table, int column);
Prototype: void algo_square_reverse(int **table, int square);

Take your previous exercice and reverse the algorithms.
Adding this piece of code to the previous one
Example :

```
// ...
algo_square_reverse(table, 0);
algo_column_reverse(table, 0);
algo_line_reverse(table, 0);
// ...
```

give you the following output :

```
Rotate Counter Clockwise square 0.
-----------
| 0 | 1 | 1 | 0 |
-----------
| 2 | 0 | 1 | 1 |
-----------
| 2 | 2 | 3 | 3 |
-----------
| 0 | 2 | 3 | 3 |
-----------

Rotate Down column 0.
-----------
| 0 | 1 | 1 | 0 |
-----------
| 0 | 0 | 1 | 1 |
-----------
| 2 | 2 | 3 | 3 |
-----------
| 2 | 2 | 3 | 3 |
-----------

Rotate Right line 0.
-----------
| 0 | 0 | 1 | 1 |
-----------
| 0 | 0 | 1 | 1 |
-----------
| 2 | 2 | 3 | 3 |
-----------
| 2 | 2 | 3 | 3 |
-----------
```

Well, your right, it's the first square, we just sorted it.

Exercise 4

Is in (2 pts)

Turn in: pool_c_d09/ex_04/is_in.c
Prototype: int is_in_line(int **table, int line, int value);
Prototype: int is_in_col(int **table, int column, int value);

Write two similar functions 'is_in_col' and 'is_in_line'.
They are taking the table as first argument, the column/line number as second argument and the value you are looking for as the last argument.
Your functions should return 0 if the given value is in the column/line, 1 otherwise.
Adding this piece of code to the previous one
Example :

```
  //...
  printf("%d - %d\n", is_in_line(table, 0, 1), is_in_col(table, 2, 3));
  printf("%d - %d\n", is_in_line(table, 3, 1), is_in_col(table, 2, 0));
  //...
```

give you the following output :

```
[...]
0 - 0
1 - 1
[...]
```

Exercise 5

Looking For Space (2 pts)

Turn in: pool_c_d09/ex_06/looking_space.c
Prototype: int *look_for_space(int **table, int *lines, int *columns, int value);

Now, you will look for an empty space in the square.

What is an empty space ? An empty space is an index of the array different value from the one we are looking for.

This function is a bit difficult to understand. We want to find a empty spot in the square, but not in the whole square. To that goal we will give the function two arrays one for lines and one for columns.

- The first parameter is the table.
- The second parameter is an array of size four. Each index refer to the a line. If a row is filled with 0, the refered line is open for search. Else if the row is filled with another value, the function should consider this line blocked and should not look into it.
- The third parameter act same as the second one but for columns.
- The last parameter is the value, so this value should not be considered as an empty space. Every other values are considered an empty space.

The return of this function is an array containing, respectivly, the line and the column where the space was found. A NULL value mean that there is no space in the given range.
In a few words, you should return the position of the first number which differ with the given one.
Example :

```
//...
#include <stdlib.h>
#define EMPTY 0
#define BLOCKED 1
//...
void    verif_return(int *ret)
{
  if (ret != NULL)
    printf("line :\t%d\nColumn :\t%d\n", ret[0], ret[1]);
  else
    printf("Nothing found in the given range.\n");
}

int     main()
{
  //...
  int   lines[4];
  int   columns[4];
  //...
  lines[0] = BLOCKED;
  lines[1] = BLOCKED;
  lines[2] = EMPTY;
  lines[3] = BLOCKED;
  columns[0] = EMPTY;
  columns[1] = EMPTY;
  columns[2] = BLOCKED;
  columns[3] = BLOCKED;
  verif_return(look_for_space(table, lines, columns, 1));
  verif_return(look_for_space(table, lines, columns, 2));
  //...
}
```

With the following input square :

```
-----------------
| 2 | 1 | 3 | 0 |
-----------------
| 0 | 0 | 1 | 1 |
-----------------
| 2 | 2 | 3 | 0 |
-----------------
| 2 | 3 | 1 | 3 |
-----------------
```

give you the following output :

```
[...]
line :  2
column: 0
Nothing found in the given range.
[...]
```

Exercise 6

Looking For Value (2 pts)

Turn in: pool_c_d09/ex_06/looking_value.c
Prototype: int *look_for_value(int **table, int *lines, int *columns, int value);

In the previous exercice, you write a function which found an empty slot. Now, write a function looking for a given value.
The parameters and the return value act same as the ones from the previous exercice.
Example :

```
//...
#include <stdlib.h>
#define EMPTY 0
#define BLOCKED 1
//...
void    verif_return(int *ret)
{
  if (ret != NULL)
    printf("line :\t%d\nColumn :\t%d\n", ret[0], ret[1]);
  else
    printf("Nothing found in the given range.\n");
}

int     main()
{
  //...
  int   lines[4];
  int   columns[4];
  //...
  lines[0] = BLOCKED;
  lines[1] = BLOCKED;
  lines[2] = EMPTY;
  lines[3] = BLOCKED;
  columns[0] = EMPTY;
  columns[1] = EMPTY;
  columns[2] = BLOCKED;
  columns[3] = BLOCKED;
  verif_return(look_for_value(table, lines, columns, 2));
  verif_return(look_for_value(table, lines, columns, 1));
  //...
}
```

With the following input square :

```
_____________
| 2 | 1 | 3 | 0 |
_____________
| 0 | 0 | 1 | 1 |
_____________
| 2 | 2 | 3 | 0 |
_____________
| 2 | 3 | 1 | 3 |
_____________
```

give you the following output :

```
[...]
line :  2
column: 0
Nothing found in the given range.
[...]
```

Exercise 7

Rotate (2 pts)

Turn in: pool_c_d09/ex_07/rotate.c
Prototype: void rotate_lines(int **table, int line, int offset);
Prototype: void rotate_columns(int **table, int column, int offset);

Those function must rotate the given column/line (given as second argument) depending of the offset given as third argument.
If the offset is negative, you should use the common algorithm else, use the reverse one.
Given the following input square :

```
_____________
| 0 | 2 | 0 | 0 |
_____________
| 0 | 3 | 1 | 3 |
_____________
| 2 | 2 | 1 | 3 |
_____________
| 1 | 1 | 3 | 2 |
_____________
```

Using this piece of code :

```
int      main()
{
  int    **table;
  //...
  int    *ret_value;
  int    *ret_space;
  int    lines[4];
  int    columns[4];
  //...
  lines[0] = BLOCKED;
  lines[1] = EMPTY;
  lines[2] = BLOCKED;
  lines[3] = BLOCKED;
  columns[0] = EMPTY;
  columns[1] = EMPTY;
  columns[2] = EMPTY;
  columns[3] = EMPTY;

  ret_value = look_for_value(table, lines, columns, 0);

  lines[0] = EMPTY;
  lines[1] = BLOCKED;

  ret_space = look_for_space(table, lines, columns, 0);

  rotate_lines(table, ret_value[0], ret_value[1] - ret_space[1]);
  print_tab(table);
  rotate_columns(table, ret_space[1], ret_value[0] - ret_space[0]);
  print_tab(table);
  //...
}
```

will produce the following output :

```
| 0 | 2 | 0 | 0 |
| 3 | 0 | 3 | 1 |
| 2 | 2 | 1 | 3 |
| 1 | 1 | 3 | 2 |

| 0 | 0 | 0 | 0 |
| 3 | 2 | 3 | 1 |
| 2 | 1 | 1 | 3 |
| 1 | 2 | 3 | 2 |
```

Exercise 8

Build Your First Square (2 pts)

Turn in: pool_c_d09/ex_09/build_first.c
Prototype: void build_first_line(int **table);
Prototype: void line_to_square(int **table, int line);

```c
#define PRINT_SQUARE_DEBUG__ 1
// ...
void    build_first_square(int *table)
{
  print_tab(table)
  build_first_line(table);
  line_to_square(table, 0);
}
```

With all those functions, you are now able to build your first square. This is the top-left square (square 0).
So, the function 'build_first_line' will use all those function to align a value on the first line (line 0).

The function 'line_to_square' will transmute the given line to a square with the same number.

Do not turn-in the function 'build_first_square', the moulinette will include it itself.

Using the output from the previous exercice and the previous piece of code, it will give you the following output :

```
_____________
| 0 | 0 | 0 | 0 |
_____________
| 3 | 2 | 3 | 1 |
_____________
| 2 | 1 | 1 | 3 |
_____________
| 1 | 2 | 3 | 2 |
_____________

Rotate Left square 0.
_____________
| 0 | 2 | 0 | 0 |
_____________
| 0 | 3 | 3 | 1 |
_____________
| 2 | 1 | 1 | 3 |
_____________
| 1 | 2 | 3 | 2 |
_____________

Rotate Left square 0.
_____________
| 2 | 3 | 0 | 0 |
_____________
| 0 | 0 | 3 | 1 |
_____________
| 2 | 1 | 1 | 3 |
_____________
| 1 | 2 | 3 | 2 |
_____________

Rotate Left line 0.
_____________
| 3 | 0 | 0 | 2 |
_____________
| 0 | 0 | 3 | 1 |
_____________
| 2 | 1 | 1 | 3 |
_____________
| 1 | 2 | 3 | 2 |
_____________

Rotate Left line 0.
_____________
| 0 | 0 | 2 | 3 |
_____________
| 0 | 0 | 3 | 1 |
_____________
| 2 | 1 | 1 | 3 |
_____________
| 1 | 2 | 3 | 2 |
_____________
```

Exercise 9

Build Your Second Square (2 pts)

Turn in: pool_c_d09/ex_09/build_second.c
Prototype: void build_last_line(int **table);

```
#define PRINT_SQUARE_DEBUG__ 0
//...
void    build_second_square(int *table)
{
  build_last_line(table);
  line_to_square(table, 3);
  push_it_up(table);
  print_tab(table);
}
```

You should now be able to build the last line of the square. And, consequently, the last square.
You should push it next to the first one. Doing this will help you build the two last one.

```
_____________
| 0 | 0 | 3 | 3 |
_____________
| 0 | 0 | 3 | 3 |
_____________
| 2 | 1 | 1 | 2 |
_____________
| 1 | 2 | 2 | 1 |
_____________
```

Exercise 10

Finish It ! (3 pts)

Turn in: pool_c_d09/ex_10/finish.c
Prototype: void build_final_line(int **table);

Now, you have to tweak the two last lines to build the complete square.
Do it as you want, you can build a line and transmute it or directly build the square.
Enjoy =D