



# C1 - Pool C

---

C-CPE-042

## Day 09-10

---

Rubik's Square



# Day 09-10

delivery method: Github  
repository name: pool\_c\_d09  
repository rights: ramassage-tek



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

- Instructions: Before You Go Further
- Introduction
- Exercise 1: Print And Check (1pts)
- Exercise 2: Algorithms (2pts)
- Exercise 3: Reverse Algorithm (2pts)
- Exercise 4: Is in (2pts)
- Exercise 5: Looking For Space (2pts)
- Exercise 6: Looking For Value (2pts)
- Exercise 7: Rotate (2pts)
- Exercise 8: Build Your First Square (2pts)
- Exercise 9: Build Your Second Square (2pts)
- Exercise 10: Finish It! (3pts)



## + 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 exercises. 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 O.
- Do not 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 algorithm, it's not a rough 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 your header file (`rubiks.h`) up to date.



This is not an easy exercise, but if you follow the exercises and ask the geckos for help, you might succeed.



## + EXERCISE 1: PRINT AND CHECK (1PTS)

Trun in: pool\_c\_d09/ex\_01/ruler.c, pool\_c\_d09/ex\_01/rubiks.h

Prototype: void print\_tab(int \*\*table);

Write a function 'print\_tab' taking a pointer of pointer on integers as parameter and returning nothing. This parameter is a four columns by four row table. You have to allocate it (with 'malloc()') in your main().

This function will produce the following output:

```
Terminal
~/C-CPE-042> gcc *.c -o print_tab
~/C-CPE-042> ./print_tab
-----
| 0 | 0 | 1 | 1 |
-----
| 0 | 0 | 1 | 1 |
-----
| 2 | 2 | 3 | 3 |
-----
| 2 | 2 | 3 | 3 |
-----

~/C-CPE-042>
```



Take care, the cosmetic around the square is composed of dash and pipes.



## + EXERCISE 2: ALGORITHMS (2PTS)

Turn in: pool\_c\_d09/ex\_02/\*, pool\_c\_d09/ex\_02/algo.c

Prototypes:

- void algo\_line(int \*\*table, int line);
- void algo\_column(int \*\*table, int column);
- 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\_. If this define is set to 1, you have to display the rotation way and the square, otherwise you shouldn't.



In your delivery, you must set the PRINT\_SQUARE\_DEBUG\_ to 0 ; to avoid the tests to fail on your debug messages.

Example:

```
Terminal
~/C-CPE-042> cat main.c
#define PRINT_SQUARE_DEBUG_ 0
// ...
int main()
{
    int **table;
    // ...
    algo_line(table, 0);
    algo_column(table, 0);
    algo_square(table, 0);
    // ...
}
// ...
~/C-CPE-042>
```



This piece of code will produce the following output:

```
Terminal
~/C-CPE-042> gcc *.c -o algo
~/C-CPE-042> ./algo
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 |
-----

~/C-CPE-042>
```



### + EXERCISE 3: REVERSE ALGORITHM (2PTS)

Turn in: pool\_c\_d09/ex\_03/\*, pool\_c\_d09/ex\_03/algo\_rev.c

Prototypes:

- void algo\_line\_reverse(int \*\*table, int line);
- void algo\_column\_reverse(int \*\*table, int column);
- void algo\_square\_reverse(int \*\*table, int square);

Take your previous exercise and reverse the algorithms.  
Adding this piece of code to the previous one

Example:

```
Terminal
~/C-CPE-042> cat main.c
// ...
    algo_square_reverse(table, 0);
    algo_column_reverse(table, 0);
    algo_line_reverse(table, 0);
// ...
~/C-CPE-042>
```

give you the following output:



```
Terminal
~/C-CPE-042> gcc *.c -o rev_algo
~/C-CPE-042> ./rev_algo
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 |
-----

~/C-CPE-042>
```

Well, you're right, it's the first square, we just sorted it!





## + EXERCISE 4: IS IN (2PTS)

Turn in: pool\_c\_d09/ex\_04/\*, pool\_c\_d09/ex\_04/is\_in.c

Prototypes:

- `int is_in_line(int **table, int line, int value);`
- `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:

```
Terminal
~/C-CPE-042> cat main.c
// ...
    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));
// ...
~/C-CPE-042>
```

Give you the following output:

```
Terminal
~/C-CPE-042> gcc *.c -o is_in
~/C-CPE-042> ./is_in
[...]
0 - 0
1 - 1
[...]
~/C-CPE-042>
```



## + EXERCISE 5: LOOKING FOR SPACE (2PTS)

Turn in: pool\_c\_d09/ex\_05/\*, pool\_c\_d09/ex\_05/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 a cell that have a 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 refers to a line. If a row is filled with 0, the referred 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, respectively, the line and the column where the space was found. A **NULL** value means 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:

```
Terminal
~/C-CPE-042> cat main.c
// ...
#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));
    // ...
}
~/C-CPE-042>
```

With the following input square:



```
Terminal
~/C-CPE-042> ./print_tab
-----
| 2 | 1 | 3 | 0 |
-----
| 0 | 0 | 1 | 1 |
-----
| 2 | 2 | 3 | 0 |
-----
| 2 | 3 | 1 | 3 |
-----

~/C-CPE-042>
```

Give you the following output:

```
Terminal
~/C-CPE-042> gcc *.c -o look_space
~/C-CPE-042> ./look_space
[...]
line : 2
column : 0
Nothing found in the given range.
[...]
~/C-CPE-042>
```



## + EXERCISE 6: LOOKING FOR VALUE (2PTS)

Turn in: pool\_c\_d09/ex\_06/\*, pool\_c\_d09/ex\_06/looking\_value.c

**Prototype:** int \*look\_for\_value(int \*\*table, int \*lines, int \*columns, int value);

In the previous exercise, you wrote 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 previous exercise.

*Example:*

```
Terminal
~/C-CPE-042> cat main.c
// ...
#include <stdlib.h>
#define EMPTY 0
#define BLOCKED 1
// ...
void verif_return (int *ret)
{
    if (ret != NULL)
        printf("line : %d\nColumn : %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:



```
Terminal
~/C-CPE-042> ./print_tab
-----
| 2 | 1 | 3 | 0 |
-----
| 0 | 0 | 1 | 1 |
-----
| 2 | 2 | 3 | 0 |
-----
| 2 | 3 | 1 | 3 |
-----

~/C-CPE-042>
```

Give you the following output:

```
Terminal
~/C-CPE-042> gcc *.c -o look_value
~/C-CPE-042> ./look_value
[...]
line : 2
column : 0
Nothing found in the given range.
[...]
~/C-CPE-042>
```



## + EXERCISE 7: ROTATE (2PTS)

Turn in: pool\_c\_d09/ex\_07/\*, pool\_c\_d09/ex\_07/rotate.c

Prototypes:

- void rotate\_lines(int \*\*table, int line, int offset);
- void rotate\_columns(int \*\*table, int column, int offset);

Those functions must rotate the given column/line (given as second parameter) depending on the offset given as third parameter.

If the offset is positive, you should use the common algorithm else, use the reverse one.

Given the following input square:

```
Terminal
~/C-CPE-042> ./print_tab
-----
| 0 | 2 | 0 | 0 |
-----
| 0 | 3 | 1 | 3 |
-----
| 2 | 2 | 1 | 3 |
-----
| 1 | 1 | 3 | 2 |
-----
~/C-CPE-042>
```

Using this piece of code:



```
Terminal
~/C-CPE-042> cat main.c
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_table(table);
    rotate_columns(table, ret_space[1], ret_value[0] - ret_space[0]);
    print_tab(table);
    // ...
}
~/C-CPE-042>
```

Will produce the following output:





```
Terminal
~/C-CPE-042> gcc *.c -o rotate
~/C-CPE-042> ./rotate
-----
| 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 |
-----

~/C-CPE-042>
```



## + EXERCISE 8: BUILD YOUR FIRST SQUARE (2PTS)

Turn in: pool\_c\_d09/ex\_08/\*, pool\_c\_d09/ex\_08/build\_first.c

**Prototypes:**

- void build\_first\_line(int \*\*table);
- void line\_to\_square(int \*\*table, int line);

```
Terminal
~/C-CPE-042> cat build_first_square.c
#define PRINT_SQUARE_DEBUG_ 0
// ...
void build_first_square(int **table)
{
    print_tab(table);
    build_first_line(table);
    line_to_square(table, 0);
}
~/C-CPE-042>
```

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 'build\_first\_square', the moulinette will include it itself.

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



```
Terminal
~/C-CPE-042> gcc *.c -o build_first
~/C-CPE-042> ./build_first
-----
| 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 |
-----

~/C-CPE-042>
```



```
Terminal
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 |
-----

~/C-CPE-042>
```



## + EXERCISE 9: BUILD YOUR SECOND SQUARE (2PTS)

Turn in: pool\_c\_d09/ex\_09/\*, pool\_c\_d09/ex\_09/build\_second.c

Prototype: void build\_last\_line(int \*\*table);

```
Terminal
~/C-CPE-042> cat build_second_square.c
#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);
}
~/C-CPE-042>
```

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.

```
Terminal
~/C-CPE-042> gcc *.c -o build_second
~/C-CPE-042> ./build_second
-----
| 0 | 0 | 3 | 3 |
-----
| 0 | 0 | 3 | 3 |
-----
| 2 | 1 | 1 | 2 |
-----
| 1 | 2 | 2 | 1 |
-----
~/C-CPE-042>
```

## + EXERCISE 10: FINISH IT! (3PTS)

Turn in: pool\_c\_d09/ex\_10/\*, 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