



# miniRE Dash

*Summary: THIS document is the subject for the miniRE Dash @42seoul.eduthon*

*version: 1.0*

# Contents

<b>I</b>	<b><u>Instruction</u></b>	<b>2</b>
<b>II</b>	<b><u>Foreword</u></b>	<b>3</b>
<b>III</b>	<b><u>Exercice 00 : Eleven number checker</u></b>	<b>5</b>
<b>IV</b>	<b><u>Exercise 01 : Phone number checker</u></b>	<b>7</b>
<b>V</b>	<b><u>Exercice 02 : Bonus : Simple E-mail validator</u></b>	<b>9</b>
<b>VI</b>	<b><u>Exercice 03 : Bonus : Push swap instruction validator</u></b>	<b>11</b>
<b>VII</b>	<b><u>Exercice 04 : Bonus : Snake to Camel</u></b>	<b>13</b>

# Chapter I

## Instructions

- Your project must be written in accordance with the Norm. If you have bonus files/functions, they are included in the norm check and you will receive a 0 if there is a norm error inside.
- Your functions should not quit unexpectedly (segmentation fault, bus error, double free, etc) apart from undefined behaviors. If this happens, your project will be considered non functional and will receive a 0 during the evaluation.
- All heap allocated memory space must be properly freed when necessary. No leaks will be tolerated.
- In this subject, you will practice how to find the **pattern** you want with *regular expressions*.
- If pattern is valid and matched with some text, you should write them in STDOUT.
- The turn in file must be a single, compilable file.
- The input value of the test cases are limited to the printable ASCII code.
- If input is NULL, print nothing.

# Chapter II

## Foreword

To find the MAC address, many pisciners use the following command

```
$>ifconfig | grep "ether" | cut -d " " -f 2
```

but, how about this?

```
$>ifconfig | grep -Eo "([0-9a-z]{2}:){5}[0-9a-z]{2}"
```

Only use grep command :)

# Chapter III

## Exercice 00 : Eleven number checker



### Exercise 00

Turn-in directory : *ex00/*

File to turn in : `eleven_number_checker.c`

Allowed functions : `write`

- Create a function that check is an input string valid *simple phone number*.
- *Simple phone number* does not contain a country code, hyphen or anything, except number. JUST plane 11 digits of number.
- Here's how it should be prototyped :

```
void    eleven_number_checker(const char *str);
```

- Running following code,


```
int main(void)
{
    eleven_number_checker((void *)0);
    eleven_number_checker("01000000000");
    eleven_number_checker("aaa01011111111aaa");
    eleven_number_checker("aaa0102222222222");
    eleven_number_checker("  01033333333  ");
    eleven_number_checker("01044A44444");
    eleven_number_checker("01055 55555");
    eleven_number_checker("01066_66666");
    return (0);
}
```

- The output must be:

```
010000000000
010111111111
010222222222
010333333333
K0
K0
K0
```

# Chapter IV

## Exercice 01 : Phone number checker

	Exercise 01
Turn-in directory : <i>ex01/</i>	
File to turn in : <code>phone_number_checker.c</code>	
Allowed functions : <code>regcomp</code> , <code>regex</code> , <code>regerror</code> , <code>regfree</code> , <code>write</code>	

- By using functions above, rewrite the function that check input number is ***valid phone number***.
- We provide example code with basic usage of Regex functions.
- In Regex manner, ***valid phone number*** starts with 3 digits of 01[0-9], followed by 3,4 digits number, and ends with 4 digits, a hyphen in between each.
- Here's how it should be prototyped :

```
void    phone_number_checker(const char *str);
```

- Running following code,

```
int main(void)
{
    phone_number_checker((void *)0);
    phone_number_checker("000-0000-0000");
    phone_number_checker("010-0000-0000");
    phone_number_checker("0100000-0000");
    phone_number_checker("010-111-1111");
    phone_number_checker("010-2222-2222010-2222-2222");
    phone_number_checker("010-3333-3333 010-3333-3333");
    phone_number_checker("010-4444-4444 010-4444-4A44");
    phone_number_checker(" 010-555-5555 010-5555-5555 ");
    phone_number_checker("abc010-6666-6666defg010-6666-6666hi jkl");
    phone_number_checker("010-7777-7010-7777-7777");
    return (0);
}
```

- The output must be:


```
K0
010-0000-0000
K0
010-111-1111
010-2222-2222
010-2222-2222
010-3333-3333
010-3333-3333
010-4444-4444
010-555-5555
010-5555-5555
010-6666-6666
010-6666-6666
010-7777-7010
```



# Bonus

## Chapter V

### Exercice 02 : Simple Password validator

	Exercise 02
Turn-in directory : <i>ex02/</i>	
File to turn in : <i>simple_password_validator.c</i>	
Allowed functions : <i>regcomp, regex, regerror, regfree, write</i>	

- Write a function to check if the input value follows our ***Password policy***.
- ***Password*** should not contain '4', '2', 'S', 'e', 'o', 'u', 'l', '.' or ' ' (space).
- Here's how it should be prototyped :

```
void    simple_password_validator(const char *str);
```



find caret, not **carrot**.

- Running following code,

```
int main(void)
{
    simple_password_validator((void *)0);
    simple_password_validator("banana31@");
    simple_password_validator("banana31.");
    simple_password_validator("apple31@");
    simple_password_validator("banana42@");
    simple_password_validator("banana31 @");
    simple_password_validator("banana31@@@@");
    return (0);
}
```


- The output must be:

```
banana31@
KO
KO
KO
KO
KO
```

# Chapter VI

## Exercise 03 :

### push\_swap instruction validator

	Exercise 03
Turn-in directory : <i>ex03/</i>	
File to turn in : <i>push_swap_instruction_validator.c</i>	
Allowed functions : <i>regcomp, regex, regerror, regfree, write</i>	

- "push\_swap" has 11 operations, "pa, pb, sa, sb, ss, ra, rb, rr, rra, rrb, rrr".
- Write a function to check instructions are valid. We will test your code with random operations, some of them are invalid form.
- Here's how it should be prototyped :

```
void    pushswap_instruction_validator(const char *str);
```



**This is a pipe and, yes. This is a hint.**

- Running following code,


```
int main(void)
{
    pushswap_instruction_validator((void *)0);
    pushswap_instruction_validator("sa");
    pushswap_instruction_validator("pb");
    pushswap_instruction_validator("rrr");
    pushswap_instruction_validator("    sa    ");
    pushswap_instruction_validator("pp");
    pushswap_instruction_validator("aa");
    pushswap_instruction_validator("sr");
    pushswap_instruction_validator("SR");
    return (0);
}
```

- The output must be:

```
sa
pb
rrr
KO
KO
KO
KO
KO
```

# Chapter VII

## Exercise 04 : Snake to Camel

	Exercise 04
snake_to_camel	
Turn-in directory : <i>ex04/</i>	
File to turn in : snake_to_camel.c	
Allowed functions : regcomp, regex, regerror, regfree, write	

- Write a function to replace a valid *snake case* input with *camel case*.
- Input string might be wrong. You have to check input is valid snake case.
- *Camel case* always starts with uppercase letter.
- Here's how it should be prototyped :

```
void    snake_to_camel(const char *str);
```



I think everyone knows what a snake case and a camel case are, but in case anyone doesn't, I prepared this.

- Running following code,

```
int main(void)
{
    snake_to_camel("");
    snake_to_camel("hello");
    snake_to_camel("hello_");
    snake_to_camel("_hello");
    snake_to_camel("_hello_");
    snake_to_camel("world_hello");
    snake_to_camel("world_hello ");
    snake_to_camel(" world_hello");
    snake_to_camel("World_hello");
    snake_to_camel("world_hello");
    snake_to_camel("world__hello");
    snake_to_camel("foo_bar world_hello");
    snake_to_camel("_");
    snake_to_camel("a_B");
    snake_to_camel("a_bc_def_ghi");
    snake_to_camel("worl d_hello");
    snake_to_camel("world_Hello");
    snake_to_camel("world_hello\n");
    snake_to_camel("42_Hello");
    snake_to_camel("world()_hi");
    return (0);
}
```

- The output must be:

```
not snake case
snake pattern
Hello
not snake case
not snake case
not snake case
snake pattern
WorldHello
not snake case
not snake case
not snake case
not snake case
not snake case
not snake case
not snake case
not snake case
snake pattern
ABcDefGhi
not snake case
not snake case
not snake case
not snake case
not snake case
```