

# Regex Dash

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

version: 1.0

# **Contents**

I	Instructions				
II /	Foreword	4			
Ш	Tips	5			
IV	Exercise 00 : Eleven number checker				
V	Exercise 01 : Phone number checker	8			
VI	Exercise 02 : Bonus : Simple E-mail validator	10			
VII	Exercise 03: Bonus: Push swap instruction validator	12			
VIII	Exercise 04 : Bonus : Snake to Camel	14			

# 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:)



If you use Character class, MUST follow ASCII definitions.

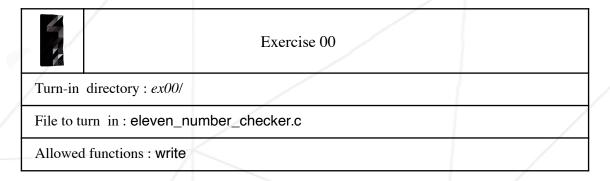
# Chapter III Tips

- Reference sites
  - Regex wiki POSIX basic and extended
  - Regexr Test regex expression realtime, rich examples and features.
- Cheat sheet

{n,m}	repeat at least n, no more then m	a{2,3}	<mark>aa</mark> , <mark>aaa</mark> , ab	Basic concept - Quantification
[n-m]	range ascii code n to m	[a-c] or [abc]	<mark>a, b, c</mark> , d, A, B, C	Meta character - Bracket[]
[^n-m]	range, but excluded	[^abd]	a b <mark>c</mark> d <mark>A B C</mark>	Meta character - Caret in Bracket[^ ]
n l m	match either n and m	[abc]l[dAB]	<mark>a b c d</mark> e f <mark>A B</mark> C D E F	Meta character - Choice

### **Chapter III**

#### Exercise 00: Eleven number checker



- Create a function that find and print 'simple phone number' pattern matched.
- *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);
```

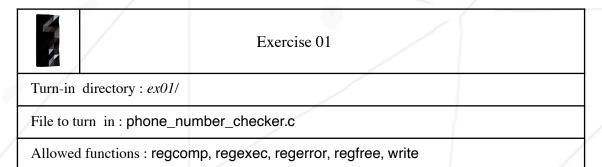
• Running following code,

```
int main(void)
{
    eleven_number_checker(NULL);
    eleven_number_checker("010000000000");
    eleven_number_checker("aaa010111111111aaa");
    eleven_number_checker("aaa010222222222222");
    eleven_number_checker(" 010333333333 ");
    eleven_number_checker("01044A44444");
    eleven_number_checker("01055 55555");
    eleven_number_checker("01066_66666");
    return (0);
}
```

```
01000000000
0101111111
0102222222
01033333333
KO
KO
```

## Chapter IV

#### Exercise 01: Phone number checker



- By using functions above, rewrite the function that find and print 'phone number' pattern matched.
- In Regex manner, *phone number* starts with 3 digits of 010, 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);



We provide example code in "{root\_dir}/regex\_example.c" with basic usage of Regex functions.

```
int main(void)
{
    phone_number_checker(NULL);
    phone_number_checker("000-0000-0000");
    phone_number_checker("010-0000-0000");
    phone_number_checker("010-0000-0000");
    phone_number_checker("010-111-1111");
    phone_number_checker("010-2222-2222");
    phone_number_checker("010-3333-3333 ");
    phone_number_checker("010-4444-4444 ");
    phone_number_checker(" 010-555-5555 ");
    phone_number_checker("abc010-6666-6666defg");
    phone_number_checker("010-7777-7010-7777-7777");
    return (0);
}
```

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

# Bonus Chapter V

#### Exercise 02: Simple Password validator



Exercise 02

Turn-in directory: ex02/

File to turn in: simple\_passward\_validator.c

Allowed functions: regcomp, regexec, regerror, regfree, write

- Write a function to check if the input value follows our *Password policy*.
- If is valid input, print it.
- 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);



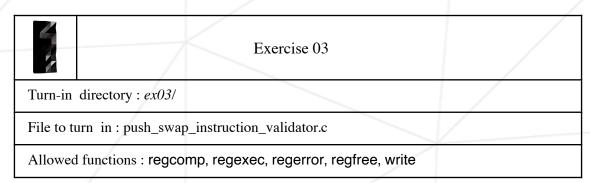
```
int main(void)
{
    simple_password_validator((void *)0);
    simple_password_validator("banana31@");
    simple_password_validator("banana31.");
    simple_password_validator("apple31@");
    simple_password_validator("banana42@");
    simple_password_validator("banana31 @");
    simple_password_validator("banana31@@@@@");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

```
banana31@
K0
K0
K0
K0
K0
K0
```

### **Chapter VI**

#### Exercise 03:

#### push\_swap instruction validator



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

void push\_swap\_instruction\_validator(const char \*str);





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

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

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

## **Chapter VII**

#### **Exercise 04: Snake to Camel**

	Exercise 04	
	snake_to_camel	
Turn-in director		
File to turn in:		
Allowed function	as : regcomp, regexec, 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 <u>snake case</u> and a <u>camel case</u> are, but in case anyone doesn't, I prepared this.

```
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);
```

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