C++ Pool - d01

C, Life, the Universe and everything else

Koalab koala@epitech.eu

Abstract: *This document is the subject for d01.*

Contents

Chapter I

Foreword

All the exercises for each day of the pool will be evaluated on the official dump (OpenSuse 12.2 GCC 4.7.2). The Koala team rejects all responsibility in case of any correction problem if you do not use the same system.
If your system is not appropriate, contact the Bocal as soon as possible so they can provide you with the required system.

For the whole duration of this pool, your exercises must be turned in using the turn-in system put in place at Epitech. The final gathering will be made at the time of end of the activities, the Epitech NTP server being the word of law on this one.

Furthermore, please note that since you will not know when any intermediary correction might be initiated, it is your own responsibility to regularly commit your work on your repository if you wish to benefit from said intermediary correction(s).

In the unlikely event that you do not find the answer you seek in the documentation, please ask an assistant in your room.

Chapter II

Exercice 0

KOALA	Exercise : 00	points : 2
Follow the white rabbit		
Turn-in directory: `(piscine_cpp_d01)/ex00`		
Compiler: `gcc`	Compilation flags: `-Wall -Wextra -Werror -std=gnu99`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `white_rabbit.c`		
Remarks : `n/a`		
Forbidden functions : `None`		

Sitting on top of the hill, Alice was bored to death, wondering if making a chaplet of flowers was worth getting up and actually gathering said flowers. And then, suddenly, a White Rabbit with pink eyes passed by, running like a madman. This was actually not really worth a mention ; and Alice wasn't much more puzzled to hear the Rabbit mumbling: "Oh my god, Oh my god ! I'm going to be late !". However, from the moment the Rabbit pulled out a watch from the pocket of his vest, looked at the time, and ran even faster, Alice was on her feet in a heartbeat. She suddenly realized that she had never seen either a Rabbit with a vest pocket, nor a watch to pull out from there. Dying to know more, she runned through the fields in the rabbit's wake, and was lucky enough to be right on time to see him rushing into a huge burrow, under the bushes. Not a moment later was she already inside, never even wondering how the hell she would get out of there.

After drifting around for a long, long time, in the maze of the burrow's walls, Alice met a Koala whose words, more or less, were these: "Hey there ye ! Wot's you doin' here ? Lookin' fer the pink pony as well ?". Not a pink pony but a white rabbit, Alice answered. "Aaah, but I know him well, th'old rabbit friend", the Koala retorted. "I even saw him not five minutes ago ! 'Looked like he was in a hell outta an hurry, th'old rabbit friend !". Alice asked the Koala to show her the direction the Rabbit was heading. Without an hesitation, the Koala pointed to his left and blurted out : "Thatta way !!", before suddenly pausing and pointing to the opposite direction. "Err, nay... I think it wos rather thatta way...". After having pointed to a dozen of different directions, the Koala finally admitted "Hmm... Actually, I think I may well be lost." Alice was in despair. She was lost in a huge burrow, and was off the trail of the white rabbit. When he saw her in this

state, the Koala took pity on her, and told her: "Dun' worry there gal, we'll find your friend th'White Rabbit. Look what I got there." He immediatly took a 37-faced dice out of his vest pocket (yes, he also has a vest pocket) and handed it to Alice. He showed each of the faces to Alice. "This is the first face - yeh can tell cause o'the number 1 written on it. And this is face 2", and so on until reaching the 37th one.

The Koala then told Alice : "What yeh got in yer hand, it's a magic dice ! Yeh must take real care of it ! But it'll make yeh find the White Rabbit ! Now, listen well, open yer hears, I'll explain to yeh how yeh must use it. Each time yeh don't know where t'go anymore, throw thees dice. Th'result'll tell yeh which direction the White Rabbit took. Although, if the dice gives yeh a multiple of 11 and the weather is nice, y'should always take a nap - might as well enjoy the sun. 'Cause yeh can be sure the White Rabbit'll do the same. But if the weather is crap, better launch the dice again. Same thing when you wake up after the nap, 'f'course. If the weather is still nice and it tells ye to nap again, you woke up waaay too early ! If you ever get a 37, then you found the White Rabbit. Never forget that after a cup of tea, you should always go straight ahead. When you get a face that's higher than 24 and that three times this face gives you seven-times-ten-and-eight or 146, means the dice was wrong, y'should turn and head back. If the result is four or 5, go left, there's a caterpillar here that sucks on that pipe of hers all day long and makes circles with her smoke. Bit crazy she is, but quite funneh ! With a 15, straight ahead with you. When the dice says 12, ye're out of luck, that was for naught and yeh have to throw again. When th'result is 13, head to yer right. Also works with 34 or more. Left it is, if the dice says six, 17 or 28. A 23 means it's 10pm ! Time for a cup of tea. Find a table, and order a lemon tea. If you don't like lemon, green tea is fine enough. Oh right, never forget to count all your results ! When you add'em and yeh find 421, yeh found the White Rabbit. Say hi for me, while you're at it. Oh, I need to tell you : that counting the results things, that also works with three-times-hundred-and-ninety-seven at least. Whenever you get a result that you can divide by 8 and get a round result with nothing left, just head back where you came from. That number 8 is crappy, I don't like it. When the result is twice or three times 5, keep going ahead, yer on the good way ! The sum though is quite a bugger, if the dice tells you you found the White Rabbit, y'still need to do what the dice told you to do ! If it tells yeh t'go left, well y'go left and th'White Rabbit'll be there. If it tells right, then you don't go left, you go right ! Well, you got it anyway. Dun worry, everything'll be fine. Ah, still, be careful if the dice tells you sumthing between eighteen and 21, go left right away ! Otherwise you'll end up meeting the Queen of Hearts. She's completely nuts, she'd never let you leave. Really, between 18 and 21 included, left as fast as yeh can ! HEy, know what ? If you ever get a 1, look on top of yer head, means the Rabbit is here ! Got it ? Remember all that ? Y'see, th'nots so complicated." Alice was head over heels with all those numbers, but she rolled the dice and went behind the White Rabbit. While she was fading away, the Koala yelled "Ah, I forget ! Whenever y'don't know what teh do, just throw the dice again !"

Write a function name `follow_the_white_rabbit`with the following prototype :

```
1 int follow_the_white_rabbit(void);
```

This function must follow the journey of Alice. You will use random(3) to simulate the dice being rolled.

When Alice must go left, you will print on the standard output :

```
gauche
```

If she must go right, you will display:

```
droite
```

If she must keep going straight ahead, you will display:

```
devant
```

If she must head back, you will display;

```
derriere
```

Then when she finds the White Rabbit, you will display :

```
LAPIN !!!
```

The function must return the sum of all the results the dice gave until now.

Every output will be on the standard output, and will be followed by a new line.

You must provide ONE FUNCTION only. Do not provide a main function, the koalinette will take care of it. It will also take care of srandom(3), so you must NOT call it yourself.

Example :

```
$>./follow_the_white_rabbit | cat -e
gauche$
droite$
droite$
devant$
derriere$
derriere$
LAPIN !!!$
```

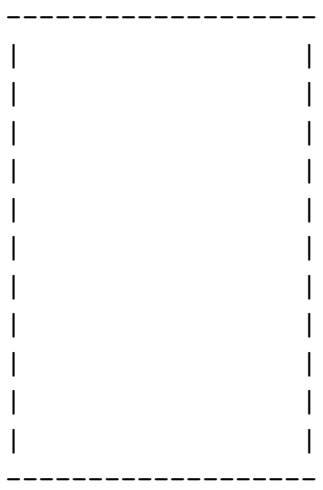
Chapter III

Exercise 1

	Exercise : 01	points : 2
The Menger Sponge		
Turn-in directory: `(piscine_cpp_d01)/ex01`		
Compiler: `gcc`	Compilation flags: `-Wextra -Werror -Wall -std=gnu99`	
Makefile: `Yes`	Rules: `all, clean, fclean, re`	
Files to turn in : `menger.c, main.c`		
Remarks : `n/a`		
Forbidden functions : `atoi`		

The Menger Sponge is a fractal curve based on squares. The idea is simple : one square is to be separated in 9, smaller, identical squares. The middle one is "empty". This process is applied to the 8 other squares.

Consider the following square (we admit this is a square) :



will become, once processed :

```
------------------
|     |     |     |
| 0,0 | 1,0 | 2,0 |
|     |     |     |
------------------
|     |     |     |
| 0,1 | 1,1 | 2,1 |
|     |     |     |
------------------
|     |     |     |
| 0,2 | 1,2 | 2,2 |
|     |     |     |
------------------
```

The (1,1) square is marked as empty and the 8 others are marked as full. The same processed is applied at each step, for each full square.

By using spaces for empty squares and # for full ones, we get :

```
Level 0
------------------
|# # #|# # #|# # #|
|# # #|# # #|# # #|
|# # #|# # #|# # #|
------------------
|# # #|# # #|# # #|
|# # #|# # #|# # #|
|# # #|# # #|# # #|
------------------
|# # #|# # #|# # #|
|# # #|# # #|# # #|
|# # #|# # #|# # #|
------------------
```

```
Level 1
------------------
|# # #|# # #|# # #|
|# # #|# # #|# # #|
|# # #|# # #|# # #|
------------------
|# # #|     |# # #|
|# # #|     |# # #|
|# # #|     |# # #|
------------------
|# # #|# # #|# # #|
|# # #|# # #|# # #|
|# # #|# # #|# # #|
------------------
```

```
Level 2
------------------
|# # #|# # #|# # #|
|#   #|#   #|#   #|
|# # #|# # #|# # #|
------------------
|# # #|     |# # #|
|#   #|     |#   #|
|# # #|     |# # #|
------------------
|# # #|# # #|# # #|
|#   #|#   #|#   #|
|# # #|# # #|# # #|
------------------
```

Write a program ‘menger‘ which, for each level, displays the size and the position of the empty square, as well as the sizes and positions of each sub-square. Every value must be displayed over 3 digits, and be separated by one space.

Your program will take as arguments the size of the original square, as well as the required number of levels.

- The size of the squares is ALWAYS a power of 3.
- The depth is ALWAYS less than, or equal to this power of 3.

```
> ./menger square_size level
```

Examples of expected outputs :

```
$> ls
Makefile main.c menger menger.c

$>./menger 3 1
001 001 001

$>./menger 9 1
003 003 003

$>./menger 9 2
003 003 003
001 001 001
001 001 004
001 001 007
001 004 001
001 004 007
001 007 001
001 007 004
001 007 007

$>./menger 27 3 | head -n 29
009 009 009
003 003 003
001 001 001
001 001 004
001 001 007
001 004 001
001 004 007
001 007 001
001 007 004
001 007 007
003 003 012
001 001 010
001 001 013
001 001 016
001 004 010
001 004 016
001 007 010
001 007 013
001 007 016
003 003 021
001 001 019
001 001 022
001 001 025
001 004 019
```

```
001 004 025
001 007 019
001 007 022
001 007 025
003 012 003
$>
```

Chapter IV

Exercise 2

	Exercise : 02	points : 2
The BMP format		
Turn-in directory: `(piscine_cpp_d01)/ex02`		
Compiler: `gcc`	Compilation flags: `-Wextra -Werror -Wall -std=gnu99`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `bitmap.h, bitmap_header.c`		
Remarks : `n/a`		
Forbidden functions : `None`		

Let's study the BMP format for a few minutes/hours.

This format is composed of three mandatory elements:

- a file header ("Bitmap file header");
- an image header ("Bitmap information header");
- the encoded image.

The file header contains 5 fields:

- a magic number the value of which must be 0x424D on 2 bytes;
- the file size on 4 bytes;
- a reserved field of two bytes, holding the value 0;
- another reserved field of 2 bytes, the value of which must be 0;

- the adress where the encoded image begins in the file (its offset).

The image header exists in many different versions. The most usual (for backward compatibility reasons) is made of 11 fields:

- the header size on 4 bytes (40 bytes in our case);
- the image's width on 4 signed bytes;
- the image's height on 4 signed bytes;
- the number of entries used in the color palette, on 2 bytes;
- the number of bits per pixel on 2 bytes (possible values are 1, 2, 4, 8, 16 and 32);
- the compression method used on 4 bytes, set to 0 if there is no compression;
- the image's size on 4 bytes (which never equals the file's size);
- the image's horizontal resolution on 4 signed bytes;
- the image's vertical resolution on 4 signed bytes;
- the size of the color palette (0 in our case) on 4 bytes;
- the number of important colors used on 4 bytes. The value should be 0 when all the colors are equally important.

Unless specified otherwise, all the fields in those headers are unsigned.

In a bitmap.h file, create two structures t_bmp_header and t_bmp_info_header, respectively representing the file header and the image header.
The `t_bmp_header` structure will have the following fields:

- magic;
- size;
- _app1;
- _app2;
- offset;

The `t_bmp_info_header` structure will have the following fields :

- size;
- width;
- height;
- planes;
- bpp;
- compression;
- raw_data_size;
- h_resolution;
- v_resolution;
- palette_size;
- important_colors.

Each of those fields will be of one of the following types:

- int16_t;
- uint16_t;
- int32_t;
- uint32_t;
- int64_t;
- uint64_t;

Those types are defined in `stdint.h`.

In a file named `bitmap_header.c,` , write two functions `make_bmp_header` and `make_bmp_info_heade`
which will initialize every member of the structures.
Since all the images we'll create will be square-shaped, the image's width will always be equal to its height.

The function `make_bmp_header` has the following signature:

```
1 void make_bmp_header(t_bmp_header * header, size_t size);
```

Its parameters are:

- header : a pointer on the `t_bmp_header` structure to initialize.
- size : the length of one of the image's sides.

The function `make_bmp_info_header` has the following signature:

```
1 void make_bmp_info_header(t_bmp_info_header * header, size_t size);
```

Its parameters are:

- header : a pointer on the `t_bmp_info_header` structure to initialize
- size : the length of one of the image's sides.

A few words about the pictures we're going to create:

- the number of entries in the color palette is always 1;
- the number of bits per pixel is always 32;
- there is no compression;
- the horizontal and vertical resolutions are always equal to 0;
- the size of the color palette is always 0;
- all the colors of our images are important;

If you are on a little endian computer (on any intel architecture, for example), the magic number's 2 bytes in t_bmp_header should have their order reversed. Indeed, on a little endian computer, any number's bytes are reversed in terms of memory representation.

The compiler always applies padding on structures, unless specified otherwise.

```
$> cat padding.c
#include <stdlib.h>
#include <stdio.h>

struct foobar
{
    char foo[2];
    int bar;
};

int main(void)

{
    printf(``%zu\n'', 2 * sizeof(char) + sizeof(int));
    printf(``%zu\n'', sizeof(struct foobar));
    return EXIT_SUCCESS;
}

$> gcc padding.c && ./a.out
6
8
$>
$>
```

The structure is bigger than the sum of the size of all of its members. The compiler has aligned the fields of the `foobar` structure. One attribute should be applied to the structure to avoid this behavior.

The attribute `packed` explicitly asks the compiler not to apply padding to the following structure.

```
$> cat padding.c
#include <stdlib.h>
#include <stdio.h>

struct __attribute__((packed)) foobar
{
    char foo[2];
    int bar;
};

int main(void)
{
    printf(``%zu\n'', 2 * sizeof(char) + sizeof(int));
    printf(``%zu\n'', sizeof(struct foobar));
    return EXIT_SUCCESS;
}
$> gcc padding.c && ./a.out
```

```
6
6
$>
$>
```

The structure's size now equals the sum of the size of its members.

Here is an example of a main function, which creates a 32x32 entirely white image:

```
$> cat main.c
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#include "bitmap.h"

int main(void)
{
    t_bmp_header header;
    t_bmp_info_header info;
    int d;
    uint32_t pixel = 0x00FFFFFF;

    make_bmp_header(&header, 32);
    make_bmp_info_header(&info, 32);

/* Not checking your return values is naugthy naughty naughty */
    d = open("32px.bmp", O_CREAT | O_TRUNC | O_WRONLY, 0644);
    write(d, &header, sizeof(header));
    write(d, &info, sizeof(info));
    for (size_t i = 0; i < 32 * 32; ++i)
        write(d, &pixel, sizeof(pixel));
    close(d);
    return EXIT_SUCCESS;
}
$> hexdump -C 32px.bmp | head -n 6
00000000 42 4d 36 10 00 00 00 00 00 00 36 00 00 00 28 00 |BM6.......6...(.|
00000010 00 00 20 00 00 00 20 00 00 00 01 00 20 00 00 00 |.. ... ..... ...|
00000020 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000030 00 00 00 00 00 00 ff ff ff 00 ff ff ff 00 ff ff |................|
00000040 ff 00 ff ff ff 00 ff ff ff 00 ff ff ff 00 ff ff |................|
*
$>
$>
```

Chapter V

Exercice 3

	Exercise : 03	points : 2
Draw me a square		
Turn-in directory: `(piscine_cpp_d01)/ex03`		
Compiler: `gcc`	Compilation flags: `-Wextra -Werror -Wall -std=gnu99`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `drawing.h, drawing.c`		
Remarks : `n/a`		
Forbidden functions : `None`		

The time has come to fill the pictures we've just created.

The image's actual content, as specified in the BMP format, can be found in the third section of the file : the encoded image.

It is stocked as a set of lines, each line following the previous one right after it. Thus, we can consider a BMP image as a two-dimensional array, with each element of this array being a pixel of our image. The origin of this array is located in the bottom-left corner of the image.

When an image has the 32 bits-per-pixel attribute, each pixel is encoded on 4 bytes (which is all good and fine). In our situation, the first byte will always equal 0. The three next byte will respectively represent the Red, Green and Blue (RGB) components of the pixel.

Here are a few examples:

```
----------------------------
| Black    |   0x00000000 |
| White    |   0x00FFFFFF |
| Red      |   0x00FF0000 |
| Green    |   0x0000FF00 |
| Blue     |   0x000000FF |
| Yellow   |   0x00FFFF00 |
----------------------------
```

In a file `drawing.h` , create a type t_point, composed of the unsigned integers. Its two fields are:

- x;
- y.

The `x` field represents the x-axis position of a point in a plane, and `y` its y-axis position.

In a file named `drawing.c` , write a function draw_square which takes a two-dimensional array representing an image as a parameter. It will draw a square of a given size to a given position.

It will have the following prototype :

```
1 void draw_square(uint32_t ** img, t_point * orig, size_t size, uint32_t
2 color);
```

Its parameters are :

- img, a two-dimensional array representing the image;
- orig, the position of the bottom-left corner of the square;
- size, the size of one of the square's sides;
- color, the color of the square to be drawn.

You will write its prototype in a file named `drawing.h`.

Here is an instance of a main function, which reuses functions from the previous exercise and generates a cyan-colored 64x64 image with a red square in the bottom-left quarter.

```
1 $> cat main.c
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <fcntl.h>
5 #include <string.h>
6
7 #include "drawing.h"
8 #include "bitmap.h"
9
```

```
10 int main(void)
11 {
12     t_bmp_header header;
13     t_bmp_info_header info;
14     unsigned int *buffer;
15     unsigned int **img;
16     t_point p = {0, 0};
17     size_t size = 64;
18     int d;
19
20     /* Creates a two-dimensional array. */
21     buffer = malloc(size * size * sizeof(*buffer));
22     img = malloc(size * sizeof(*img));
23     memset(buffer, 0, size * size * sizeof(*buffer));
24     for (size_t i = 0; i < size; ++i)
25         img[i] = buffer + i * size;
26
27     make_bmp_header(&header, size);
28     make_bmp_info_header(&info, size);
29
30     draw_square(img, &p, size, 0x0000FFFF);
31     p.x = 10;
32     p.y = 10;
33     draw_square(img, &p, 22, 0x00FF0000);
34
35     d = open("square.bmp", O_CREAT | O_TRUNC | O_WRONLY, 0644);
36     write(d, &header, sizeof(header));
37     write(d, &info, sizeof(info));
38     write(d, buffer, size * size * sizeof(*buffer));
39     close(d);
40     return EXIT_SUCCESS;
41 }
42 $>
```

Chapter VI

Exercise 4

	Exercise : 04	points : 4
Draw me a sponge		
Turn-in directory: `(piscine_cpp_d01)/ex04`		
Compiler: `gcc`	Compilation flags: `-Wextra -Werror -Wall -std=gnu99`	
Makefile: `Yes`	Rules: `all, clean, fclean, re`	
Files to turn in : `drawing.h, drawing.c, bitmap.h, bitmap_header.c, menger.c, main.c`		
Remarks : `n/a`		
Forbidden functions : `None`		

Now that you know how to create bitmap files and draw squares in them, you have all the required elements to draw a face of a Menger's Sponge.

You will write a program menger_face which generates an image of a given size, picturing the face of a Menger's Sponge at a given depth.

This program will take as arguments the name of the image file to create, the size of the image's sides, and the required depth for the Menger's Sponge. If the number of arguments is incorrect, you must return a value different than 0 and print the following message on the standard error output, followed with a newline.

```
1 menger_face file_name size level
```

The three arguments are:

- the name of the file to create;
- the size of the image's sides;
- the required depth of the Sponge.

Full squares will be black-colored, and empty squares will be grey-colored. The colors will actually be tightly related to the current depth of the Sponge. Each component of the color will be equal to 0xFF, divided by the remaining depth level plus one. Thus, the smallest empty squares of the Sponge will always be white.

A color is considered grey whenever its three components have the same value.

For a total depth of 3, we would get :

Depth	Color	
1	255 / 3	0x00555555
2	255 / 2	0x007F7F7F
3	255 / 1	0x00FFFFFF

Chapter VII

Exercise 5

	Exercise : 05	points : 3
It must be nice from up there		
Turn-in directory: `(piscine_cpp_d01)/ex05`		
Compiler: `gcc`	Compilation flags: `-Wextra -Werror -Wall -std=gnu99`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `pyramid.c`		
Remarks : `n/a`		
Forbidden functions : `None`		

You are stuck at the top of a pyramid. Each room inside leads to two neighboring rooms on the lower floor.

```
1          0
2        1 2
3      3 4 5
4    6 7 8 9
```

Thus, from room 0, one can access rooms 1 and 2. From room 2, one can reach rooms 4 and 5 and from room 4, we can go to room 7 and 8.

The only thing in your possession is the map of the pyramid you're stuck in. It displays the distance between each room.

```
1          0
2        7 4
3      2 3 6
4    8 5 9 3
```

From the top, there's 7 meters to reach the left room and only 4 to reach the room on the right.

Your goal is to find the shortest way to exit the pyramid. In our example, that would be :

```
                              0 + 4 + 3 + 5
      which equals
                                   12
```

In a file named pyramid.c, write a function named `pyramid_path` with the following prototype :

```
int pyramid_path(int size, int ** map);
```

This function returns the total distance traveled to get out of the pyramid. Its parameters are:

- size, the height of the pyramid;
- map, a two-dimensional array containing the map of the pyramid.

In the previous example, the "map" parameter would be declared as follow:

```
{
  {0},
  {7, 4},
  {2, 3, 6},
  {8, 5, 9, 3}
};
```

You must NOT provide a main function.

Here's a more complicated pyramid:

```
                                   00
                                 95 64
                               17 47 82
                             18 35 87 10
                           20 04 82 47 65
                         19 01 23 75 03 34
                       88 02 77 73 07 63 67
```

```
99 65 04 28 06 16 70 92
41 41 26 56 83 40 80 70 33
41 48 72 33 47 32 37 16 94 29
53 71 44 65 25 43 91 52 97 51 14
```

Chapter VIII

Exercise 6

	Exercise : 06	points : 3
Fook this, seriously		
Turn-in directory: `(piscine_cpp_d01)/ex06`		
Compiler: `gcc`	Compilation flags: `-Wextra -Werror -Wall -std=gnu99`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `ex_6.h`		
Remarks : `n/a`		
Forbidden functions : `None`		

You will write the file `ex_6.h` needed for the following code to compile and print the expected output.

```
$>cat main.c
#include <stdlib.h>
#include <stdio.h>

#include "ex_6.h"

int main(void)
{
    t_foo foo;

    foo.bar = 0;
    foo.foo.foo = 0xCAFE;
    printf("%d\n", sizeof(foo) == sizeof(foo.foo));
    printf("%d\n", sizeof(foo.foo.bar.foo) == sizeof(foo.foo.foo));
    printf("%d\n", sizeof(foo.bar) == 2 * sizeof(foo.foo.bar));
    printf("%d\n", sizeof(foo.foo.foo) == sizeof(foo.foo.bar.bar));
    printf("%08X\n", foo.bar);
    return EXIT_SUCCESS;
}
$>ls
ex_6.h main.c
```

```
$>gcc -Wall -Wextra -Werror -std=c99 main.c
$>./a.out
1
1
1
1
0000CAFE
$>
$>
```

You MUST NOT provide the file main.c

Chapter IX

Exercise 7

	Exercise : 07	points : 3
Koalatchi		
Turn-in directory: `(piscine_cpp_d01)/ex07`		
Compiler: `gcc`	Compilation flags: `-Wextra -Werror -Wall -std=gnu99`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `koalatchi.c`		
Remarks : `n/a`		
Forbidden functions : `None`		

We are now going to study the Koalatchi, well-known ancester of the tamagotchi.

For the uncultured swines amongst us, a Koalatchi is a virtual Koala. Its owner must take care of him, so that the Koalatchi might become an overpowered being able to take over the world.

The only problem is our underlying laziness, and raising a Koala is a long and arduous task (even when it's a virtual one). We're going to use the wonderful API (Application Programming Interface) that its creator have gifted the koalatchi with. This API, using pre-determined messages, will allow us to acknowledge and take care of our Koalatchi's needs.

Each message is composed of a 4 bytes header, and can possibly contain a string of characters.

There are three types of messages:

- Request, occurs when the Koala wants to ask something to his master, or when the master wants his Koala to perform a specific action;

- Notification, occurs when a Koala wants to inform his master about something he just did, and vice-versa;
- Error, occurs when the Koala encounters an impossible situation (which can induce various hazards such as death).

Each message has a specific domain of application. These domains are:

- Nutrition;
- Entertainment;
- Education.

However, a message can only have a single type and a single domain of application.

The message header is composed as follow:

- The message's type on 1 byte. The available values are:
  - Notification : 1
  - Request : 2
  - Error : 4
- The domain of application on 1 byte. The available values are:
  - Nutrition : 1;
  - Entertainment : 2;
  - Education : 4;
- A unique value describing the message, on 2 bytes.

Here are, for each domain, all the possible messages that can be emitted:

- Nutrition :

- Notification
  - Eat : the master feeds his Koala
    + Value of the last 2 bytes : 1
  - Defecate : the Koala defecates
    + Value of the last 2 bytes : 2
- Request
  - Hungry : the Koala is hungry
    + Value of the last 2 bytes : 1
  - Thirsty : the Koala is thirsty
    + Value of the last 2 bytes : 2
- Error
  - Indigestion : the Koala has an indigestion
    + Value of the last 2 bytes : 1
  - Starving : the Koala is starving
    + Value of the last 2 bytes : 2

- Entertainment :
  - Notification
    - Ball : le Koala plays with a ball
      + Value of the last 2 bytes : 1
    - Bite : le Koala bites his Master (how entertaining !)
      + Value of the last 2 bytes : 2
  - Request
    - NeedAffection : the Koala needs love and kindness from his Master
      + Value of the last 2 bytes : 1
    - WannaPlay : the Koala or the Maitre want to play
      + Value of the last 2 bytes : 2

        * Hug : the Master and the Koala are cuddling
          + Value of the last 2 bytes : 3

    - Error

        * Bored : the Koala is bored to death
          + Value of the last 2 bytes : 1

- Education :

    - Notification :

        * TeachCoding : the Master teaches how to code to the Koala
          + Value of the last 2 bytes : 1
        * BeAwesome : the Master teaches how to be AWESOME to the Koala
          + Value of the last 2 bytes : 2

    - Request :

        * FeelStupid : the Koala feels stupid and craves for knowledge
          + Value of the last 2 bytes : 1

    - Error :

        * BrainDamage : the Koala has such a headache he can see flamingos.
          + Value of the last 2 bytes : 1

In a file named `koalatchi.c` , write the function named `prettyprint_message` with the following signature:

```
1 int prettyprint_message(uint32_t header, char const * content);
```

Its parameters are:

- header, the message header;

- content, the message itself.

This function will display every detail of the message in a human-readable way, using the following format:

```
TYPE DOMAIN ACTION [CONTENT]
```

If the parameter `content` is `NULL` , do not print anything after the action.
If the message is valid, the function returns 0. Otherwise, it will return 1. If a message is not valid, the function must print out :

```
Invalid message.
```

You should print everything on the standard output, followed by a new line.

Using unions is FORBIDDEN.

You must use at least two of the following operators:

```
<< >> & | ^ ~
```

Here is an example of a main function:

```
$> cat main.c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

int prettyprint_message(uint32_t, char const *);

int main(void)
{
    prettyprint_message(0x00C0FFEE, "Needed !");

    prettyprint_message(0x02010001, "\"Kreog !\"");
    prettyprint_message(0x01010001, "Eucalyptus");
    prettyprint_message(0x01010002, "\"CACA !\"");
    prettyprint_message(0x01010001, "Keytronic");
    prettyprint_message(0x04010001, NULL);

    /* Dark voodoo incantations to resurect the Koala. */
```

```
19
20      prettyprint_message(0x02020001, NULL);
21      prettyprint_message(0x01040002, NULL);
22      prettyprint_message(0x01020002, "\"KREOG !!!\"");
23      prettyprint_message(0x01040001, "Brainfuck");
24      prettyprint_message(0x04040001, "\"Dark Moon of the side...\"");
25
26      return 0;
27  }
28  $> gcc -Wall -Wextra -Werror -std=gnu99 main.c koalatchi.c
29  $> ./a.out | cat -e
30  Invalid message.$
31  Request Nutrition Hungry "Kreog !"$
32  Notification Nutrition Eat Eucalyptus$
33  Notification Nutrition Defecate "CACA !"$
34  Notification Nutrition Eat Keytronic$
35  Error Nutrition Indigestion$
36  Request Entertainment NeedAffection$
37  Notification Education BeAwesome$
38  Notification Entertainment Bite "KREOG !!!"$
39  Notification Education TeachCoding Brainfuck$
40  Error Education BrainDamage "Dark Moon of the side..."$
41  $>
42  $>
```

You MUST NOT provide a main function.

Chapter X

Exercise 8

	Exercise : 08	points : 3
Log		
Turn-in directory: `(piscine_cpp_d01)/ex08`		
Compiler: `gcc`	Compilation flags: `-Wextra -Werror -Wall -std=gnu99`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `log.h, log.c`		
Remarks : `n/a`		
Forbidden functions : `None`		

Logs take a very important role in computer science. Thanks to them, one can keep a record of everything that has been performed. Taking a peek in /var/log shows the amount of informations that are being kept, either by various applications or by the operating system itself.

We are going to write down a few logging functions. They should provide a way to choose where the messages we want to log will be headed. Thus, it must be possible to print on the standard output, on the error output or even in a file of our choosing. The error output should be the default choice. If a program logs messages in a file, each message should be appended at the end of the file.

Furthermore, to each message will be associated a log level. These levels are inspired by syslog(3), and are as follow:

- Error
- Warning
- Notice
- Info

- Debug

All these levels will be defined in an enumeration named LogLevel.

It must be possible to choose the maximum desired log level.
For example, if the maximum desired level is Warning, the only messages that should actually be logged will be Error and Warning-level messages. The default behavior will default the maximum desired level to Error, thus only logging Error-level messages.

Messages will all be formatted as follow:

```
Date [Level]: Message
```

The date should have the same format as the one returned by ctime(3). You must obtain the system time with a call to time(2).

In a file named log.h, write an enum LogLevel which contains all the enumerators previously defined.

In a file named `log.c` , implement the following functions:

```
enum LogLevel get_log_level(void);
enum LogLevel set_log_level(enum LogLevel);
int set_log_file(char const *);
int close_log_file(void);
int log_to_stdout(void);
int log_to_stderr(void);
void log_msg(enum LogLevel, char const * fmt, ...);
```

The function `get_log_level` returns the current log level.

The function `set_log_level` defines the log level to be used. This level is passed as its parameter. If the requested log level does not exist, the current level is left unchanged. This function returns the current log level at the end of the function call.

The function `set_log_file` allows the user to provide the name of the file where the messages will be written. The file's name is passed as its parameter. If another file was previously opened, it will be closed beforehand. The function returns 0 upon success, 1 otherwise.

The function `close_log_file` closes the current log file (if it does exist) and resets the log ouput to the error output. If no file was previously opened, this function returns without doing anything. It returns 0 if no error was encountered, 1 otherwise.

The function `log_to_stdout` sets the log output to the standard output. If a file was previously opened, the functions closes it beforehand. It returns 0 if no error was encountered, 1 otherwise.

The function `log_to_stderr` sets the log output to the error output. If a file was previously opened, the functions closes it beforehand. It returns 0 if no error was encountered, 1 otherwise.

The function `log_msg` writes a message on the previously set log output. Its parameters are the log level of the message, a printf-like format string, and variadic arguments. If the required log level does not exist, this function returns with no further action.

The messages' destination, as well as the current log level, must be stocked in global variables that MUST NOT be accessible through functions that were not defined in the log.c compilation unit.

A few things you are advised to read: `fopen(3)`, `fprintf(3)`, `vfprintf(3)`, `ctime(3)`, `time(2)`, and `stdarg(3)`.

Here is an example of a main function:

```
$>ls
log.c log.h main.c
$>cat main.c
#include <stdio.h>
#include <stdlib.h>

#include "log.h"

int main(void)
{
    set_log_file("out.log");
    set_log_level(Debug);
    log_msg(Debug, "This is debug\n");
    log_msg(42, "This should not be printed\n");
    log_msg(Warning, "This is a warning\n");
    set_log_level(Warning);
    log_msg(Info, "This is info\n");
    log_msg(Error, "KREOG !\n");
```

```
    close_log_file();
    return EXIT_SUCCESS;
}
$>gcc -Wall -Wextra -Werror -std=c99 main.c log.c
$>./a.out
$>ls
a.out log.c log.h main.c out.log
$>cat out.log
Tue Dec 7 01:08:19 2010 [Debug]: This is debug
Tue Dec 7 01:08:19 2010 [Warning]: This is a warning
Tue Dec 7 01:08:19 2010 [Error]: KREOG !
$>./a.out && cat out.log
Tue Dec 7 01:08:19 2010 [Debug]: This is debug
Tue Dec 7 01:08:19 2010 [Warning]: This is a warning
Tue Dec 7 01:08:19 2010 [Error]: KREOG !
Tue Dec 7 01:11:09 2010 [Debug]: This is debug
Tue Dec 7 01:11:09 2010 [Warning]: This is a warning
Tue Dec 7 01:11:09 2010 [Error]: KREOG !
$>
```

You MUST NOT provide a main function.