



# B3 - C++ Pool

---

B-PAV-242

## Day 01

---

C, Life, the Universe and everything else



**KOALA**

42.0



# Day 01

binary name: no binary  
group size: 1  
repository name: cpp\_d01  
repository rights: ramassage-tek  
language: C



- Your repository must contain the totality of your source files, but no useless files (binary, temp files, obj files,...).

## FOREWORD

For the entirety of this pool, your exercises must be turned in using the turn-in system put in place by Epitech.

Intermediary tests may be executed at any time on any given day. As you do not know when these tests will be executed, it is of your own responsibility to regularly commit your work if you wish to benefit from these tests.

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



## GENERAL SETPOINTS



If you do half the exercises because you have comprehension problems, it's okay, it happens. But if you do half the exercises because you're lazy, and leave at 2PM, you **WILL** have problems. Do not tempt the devil.



Every function implemented in a header or unprotected header leads to 0 for the exercise.



All output goes to the standard output, and must be ended by a newline, unless specified otherwise.



To avoid compilation problems during automated tests, please include all necessary files within your headers.

Please note that none of your files must contain a `main` function, unless specified otherwise. We will use our own `main` functions to compile and test your code.



## UNIT TESTS

---

It is highly recommended to test your functions as you implement them. It is common practice to create and use what are called **unit tests**.

From now on, we expect you to write unit tests for your functions (when possible). To do so, please follow the instructions in the “**How to write Unit Tests**” document on the intranet, available [here](#).

Create a directory named `tests`. For each of the functions you turn in, create a file in that directory named `tests-FUNCTION_NAME.c` containing all the tests needed to cover all of the exercise’s possible cases (regular or irregular).

Here is a sample set of unit tests for the `my_strlen` function:


```
#include <riterion/criterion.h>

Test(my_strlen, positive_return_value)
{
    cr_assert_eq(my_strlen("toto"), 4);
}

Test(my_strlen, empty_string)
{
    cr_assert_eq(my_strlen(""), 0);
}
```



## EXERCISE 0 - FOLLOW THE WHITE RABBIT

	Exercise: 00		points : 2
Follow the White Rabbit			
Turn-in directory: <code>cpp_d01/ex00</code>			
Compiler: <code>gcc</code>		Compilation flags: <code>-Wall -Wextra -Werror -std=gnu99</code>	
Makefile: <code>No</code>		Rules: <code>n/a</code>	
Files to turn in: <code>white_rabbit.c</code>			
Notes: <code>None</code>			
Forbidden functions: <code>None</code>			

Sitting on top of the hill, Alice was bored to death, wondering if making a chaplet of flowers was worth getting up and gathering said flowers. And then, suddenly, a White Rabbit with pink eyes passed by, running like a madman. This wasn't really worth mentioning, and Alice wasn't so puzzled to hear the Rabbit mumbling: "Oh my god, Oh my god! I'm going to be late!"

However, as soon as the Rabbit pulled a pocket watch from its 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 a rabbit with a vest pocket, nor a watch to pull out of it.

Dying to know more, she ran through the fields in the rabbit's wake, and was lucky enough to be right on time to see it rushing into a huge burrow, under the bushes.

Not a moment later was she already inside, not 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 could be approximated to: "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 of a hurry, th'old rabbit friend!"

Alice asked the Koala to show her the direction the Rabbit was heading. Without hesitating, 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 was rather thatta way...". After having pointed to a dozen 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 here.". He immediately took a 37-sided die out of his vest pocket (yes, he also had a vest pocket) and handed it to Alice. He showed every single side to Alice. "This is the first side - yeh can tell cause o' the number 1 written on it. And this is side 2", and so on, until he reached the 37th side.

The Koala then told Alice: "What yeh got in yer hand, it's a magic die! Yeh must take real good care of it! But it'll help yeh find the White Rabbit! Now, listen well, open yer ears, I'll explain to yeh how yeh must use it. Every time yeh don't know where t'go anymore, throw the die. Theresult'll tell yeh which direction the White Rabbit took. Although, if the die 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 throw the die 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 side that's higher than 24 and that three times this side gives you seven-times-ten-and-eight or 146, means the die was wrong, y'should turn and head back. If the result is four or 5, go left, there's a caterpillar here that smokes his pipe all day long and blows smoke rings. Bit crazy he is, but quite funneh! With a 15, straight ahead with you. When the die 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 die 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 thing, 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 die tells you you found the White Rabbit, y'still need to do what the die 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 die tells you sumthing between 18 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 you can! Hey, know what? If you ever get a 1, look on top of yer head, means the Rabbit is there! 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 after the White Rabbit. While she was fading away, the Koala yelled "Ah, I forgot! Whenever y'don't know what teh do, just throw the die again!"



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

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

This function must follow Alice's journey. You will use `random(3)` to simulate the dice being rolled. When Alice must go left, print the following on the standard output:

`gauche`

If she must go right, print `droite`.

If she must keep going straight ahead, print `devant`.

If she must head back, print `derriere`.

When she finds the White Rabbit, print `LAPIN !!!`.

The function must return the sum of all the results the die has given up to this point.


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

Here is an example:

```
Terminal
~/B-PAV-242> ./follow_the_white_rabbit | cat -e
gauche$
droite$
droite$
devant$
derriere$
derriere$
LAPIN !!!$
```

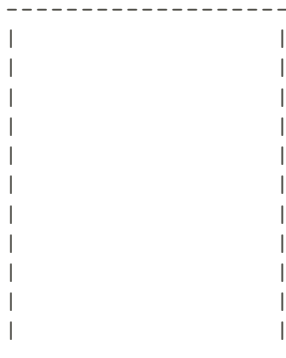


## EXERCISE 1 - THE Menger SPONGE

	Exercise: 01	points : 2
Follow the White Rabbit		
Turn-in directory: cpp_d01/ex01		
Compiler: gcc	Compilation flags: -Wall -Wextra -Werror -std=gnu99	
Makefile: Yes	Rules: all, clean, fclean, re	
Files to turn in: menger.c, main.c		
Notes: None		
Forbidden functions: atoi		

The Menger Sponge is a fractal curve based on squares. The idea is simple: one square is to be split into 9 smaller, identical squares. the middle one is "empty". This process is then repeated for the 8 other squares.

Consider the following square:







Once processed, that square will become:

-----											
	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 process is repeated each step, for each full square.

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

Level 0

```
-----
|# # #|# # #|# # #|
|# # #|# # #|# # #|
|# # #|# # #|# # #|
-----
|# # #|# # #|# # #|
|# # #|# # #|# # #|
|# # #|# # #|# # #|
-----
|# # #|# # #|# # #|
|# # #|# # #|# # #|
|# # #|# # #|# # #|
-----
```

Level 1

```
-----
|# # #|# # #|# # #|
|# # #|# # #|# # #|
|# # #|# # #|# # #|
-----
|# # #|     |# # #|
|# # #|     |# # #|
|# # #|     |# # #|
-----
|# # #|# # #|# # #|
|# # #|# # #|# # #|
|# # #|# # #|# # #|
-----
```

Level 2

```
-----
|# # #|# # #|# # #|
|#   #|#   #|#   #|
|# # #|# # #|# # #|
-----
|# # #|     |# # #|
|#   #|     |#   #|
|# # #|     |# # #|
-----
|# # #|# # #|# # #|
|#   #|#   #|#   #|
|# # #|# # #|# # #|
-----
```

Write a program called `menger` which, for each level, displays

- The size and the position of the empty square
- The sizes and positions of each sub-square.

Every value must be displayed over 3 digits, and separated by a single 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
```

Example:




```
Terminal
~/B-PAV-242> ls
Makefile main.c menger menger.c
~/B-PAV-242> ./menger 3 1
001 001 001
~/B-PAV-242> ./menger 9 1
003 003 003
~/B-PAV-242> ./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
```



```
Terminal
~/B-PAV-242> ./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
```



## EXERCISE 2 - THE BMP FORMAT

	Exercise: 02		points : 2
The BMP format			
Turn-in directory: <code>cpp_d01/ex02</code>			
Compiler: <code>gcc</code>		Compilation flags: <code>-Wall -Wextra -Werror -std=gnu99</code>	
Makefile: No		Rules: n/a	
Files to turn in: <code>bitmap.h</code> , <code>bitmap_header.c</code>			
Notes: None			
Forbidden functions: None			

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

A BMP file 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 holding the value 0, on 2 bytes
- Another reserved field holding the value 0, on 2 bytes
- The address where the encoded image begins in the file (its offset), on 4 bytes

There are many different versions of the image header. The most common (for backward compatibility reasons), is composed of 11 fields:

- The header size on 4 bytes (the header size being 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, set to 0 if there is no compression, on 4 bytes
- 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 `t_bmp_header` and `t_bmp_info_header` structures, 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 one of the following types:

- `int16_t`
- `uint16_t`
- `int32_t`
- `uint32_t`
- `int64_t`
- `uint64_t`

In a file named `bitmap_header.c`, define two `make_bmp_header` and `make_bmp_info_header` functions, 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 `make_bmp_header` function has the following signature:

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

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

The `make_bmp_info_header` function has the following signature:

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

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

A word 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 to structures, unless specified otherwise.

```
#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;
}
```

*padding.c*

```
Terminal
~/B-PAV-242> gcc padding.c && ./a.out
6
8
```

The structure is larger than the sum of the size 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 `packed` attribute explicitly tells the compiler not to apply padding to the following structure.

```
#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;
}
```

*padding.c*



```
Terminal
~/B-PAV-242> gcc padding.c && ./a.out
6
6
```

The structure's size is now equal to the sum of its member's size.

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

```
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include "bitmap.h"

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

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

    // Not checking your return values is naughty naughty naughty
    int 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;
}
```


*main.c*

```
Terminal
~/B-PAV-242> hexdump -C 32px.bmp | head -n 6
00000000 42 4d 36 10 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 |..  ...  ...  ...|
00000020 00 00 00 10 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 |.....|
```





## EXERCISE 3 - DRAW ME A SQUARE

	Exercise: 03	points : 2
Draw me a square		
Turn-in directory: cpp_d01/ex03		
Compiler: gcc	Compilation flags: -Wall -Wextra -Werror -std=gnu99	
Makefile: No	Rules: n/a	
Files to turn in: drawing.h, drawing.c		
Notes: None		
Forbidden functions: None		

It is now time to fill in the pictures you can now create.

The BMP format specifies that the image content can be found in the third section of the file: the encoded image.

It is stored as a set of lines, with no delimiter between lines. BMP images can therefore be seen as a two-dimensional array, each element of this array being a pixel of our image. Keep in mind that the first pixel in the array is the bottom-left corner of the image.

Of the 32 bits used to represent a pixel, the first byte will always be equal to 0 in our case. The three other bytes represent the Red, Green and Blue (RGB) components of the pixel.

Here are some examples:

Color	Hex
Black	0x00000000
White	0x00FFFFFF
Red	0x00FF0000
Green	0x0000FF00
Blue	0x000000FF
Yellow	0x00FFFF00



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

- `x`: the x-axis position of a point in a plane
- `y`: the y-axis position of a point in a plane

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

```
void draw_square(uint32_t **img, t_point *origin, size_t size, uint32_t color);
```

- `img`: a two-dimensional array representing the image
- `origin`: the position of the bottom-left corner of the square
- `size`: the size of the square's sides
- `color`: the color of the square to be drawn

It will be declared in the `drawing.h` file.



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 corner.

```
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include "drawing.h"
#include "bitmap.h"

int main(void)
{
    unsigned int \*buffer = malloc(size * size * sizeof(\*buffer));
    memset(buffer, 0, size * size * sizeof(\*buffer));

    unsigned int \*\*img = malloc(size * sizeof(\*img));
    for (size_t i = 0; i < size; ++i)
        img[i] = buffer + i * size;

    t_bmp_header header;
    t_bmp_info_header info;

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

    t_point p = {0, 0};
    size_t size = 64;
    draw_square(img, &p, size, 0x0000FFFF);

    p.x = 10;
    p.y = 10;
    draw_square(img, &p, 22, 0x00FF0000);


    int d = open("square.bmp", O_CREAT | O_TRUNC | O_WRONLY, 0644);
    write(d, &header, sizeof(header));
    write(d, &info, sizeof(info));
    write(d, buffer, size * size * sizeof(\*buffer));
    close(d);

    return EXIT_SUCCESS;
}
```

*main.c*



## EXERCISE 4 - DRAW ME A SPONGE

	Exercise: O4	points : 4
Draw me a sponge		
Turn-in directory: cpp_d01/exO4		
Compiler: gcc	Compilation flags: -Wall -Wextra -Werror -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		
Notes: None		
Forbidden functions: None		

You now 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 `menger_face` program which generates an image of a given size, depicting the face of a Menger's sponge at a given depth.

The 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 non-null value and print the following message on the standard error output, followed by a newline.

```
menger_face file_name size level
```

Full squares will be colored in black, and empty squares in grey. The colors will actually be tightly coupled 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 value is considered grey when its three components have the same value

A depth of 3 would produce the following values:

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



## EXERCISE 5 - IT MUST BE NICE FROM UP THERE

	Exercise: 05	points : 3
It must be nice from up there		
Turn-in directory: cpp_d01/ex05		
Compiler: gcc	Compilation flags: -Wall -Wextra -Werror -std=gnu99	
Makefile: No	Rules: n/a	
Files to turn in: pyramid.c		
Notes: None		
Forbidden functions: None		

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

```
0
1 2
3 4 5
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 rooms 7 and 8.

The only thing in your possession is the map of the pyramid you're stuck in. It indicates the distance between rooms.

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

There are 7 meters between the top level and the left room, and only 4 between the top level and the right one.

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

```
0 + 4 + 3 + 5
which is equal to
12
```

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

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

The 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 follows:

```
{  
    { 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
```



## EXERCISE 6 - FOOK THIS, SERIOUSLY

	Exercise: O6	points : 3
Fook this		
Turn-in directory: cpp_d01/exO6		
Compiler: gcc	Compilation flags: -Wall -Wextra -Werror -std=gnu99	
Makefile: No	Rules: n/a	
Files to turn in: ex_6.h		
Notes: None		
Forbidden functions: None		

Create the `ex_6.h` file needed for the following code to compile and generate the expected output.

```
#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;
}
```

*main.c*



```
Terminal
~/B-PAV-242> ls
ex_6.h main.c
~/B-PAV-242> gcc -Wall -Wextra -Werror -std=c99 main.c
~/B-PAV-242> ./a.out
1
1
1
1
0000CAFE
~/B-PAV-242>
```




You must NOT provide the `main.c` file





## EXERCISE 7 - KOALATCHI

	Exercise: 07		points : 3
Koalatchi			
Turn-in directory: <code>cpp_d01/ex07</code>			
Compiler: <code>gcc</code>		Compilation flags: <code>-Wall -Wextra -Werror -std=gnu99</code>	
Makefile: <code>No</code>		Rules: <code>n/a</code>	
Files to turn in: <code>koalatchi.c</code>			
Notes: <code>None</code>			
Forbidden functions: <code>None</code>			

We are now going to study the Koalatchi, the famous toy that inspired the Tamagotchi.

For the uncultured swine among us, a Koalatchi is a virtual Koala. Its owner must take care of it so that the Koalatchi might someday become an all-powered being and take over the world.

The only problem is that we humans are lazy, 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) provided by the Koalatchi's creators. This API lets users acknowledge and take care of a Koalatchi's needs through the use of pre-determined messages.

Each message is composed of a 4-byte header and can possibly contain a string of characters.

There are three types of messages:

- **Request:** occurs when the Koala wants something from its master, or when the master wants the Koala to perform a specific action
- **Notification:** occurs when the Koala wants to inform its master of something it did, and vice-versa
- **Error:** occurs when the Koala is faced with an impossible situation (which can lead to various hazards such as death)

Each message has a specific application domain. These domains can be:

- Nutrition
- Entertainment
- Education

The message header has the following structure:

- The message type on 1 byte. Possible values are:
  - Notification: 1
  - Request: 2
  - Error: 4
- The application domain on 1 byte. Possible values are:
  - Nutrition: 1



- Entertainment: 2
- Education: 4
- A unique value describing the message, on 2 bytes

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

- Nutrition
  - Notification
    - Eat: the master feeds the 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: the Koala plays with a ball
      - Value of the last 2 bytes: 1
    - Bite: the Koala bites its master (how entertaining!)
      - Value of the last 2 bytes: 2
  - Request
    - NeedAffection: the Koala needs love and care from its master
      - Value of the last 2 bytes: 1
    - WannaPlay: the Koala or the master want to play
      - Value of the last 2 bytes: 2
    - Hug: the Koala and the master 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 its Koala how to code
    - Value of the last 2 bytes: 1
  - BeAwesome: the master teaches its Koala how to be AWESOME
    - Value of the last 2 bytes: 2
- Request
  - FeelStupid: the Koala feels stupid and craves knowledge
    - Value of the last 2 bytes: 1
- Error
  - BrainDamage: the Koala's headache is so bad it can see flamingos
    - Value of the last 2 bytes: 1



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

```
int prettyprint_message(uint32_t header, const char *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 manner, using the following format:

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

If the `content` parameter is null, nothing should be printed after the action.

If the message is valid, the function returns 0. Otherwise, it returns 1. If a message is not valid, the function must output:

```
Invalid message.
```



The use of unions is FORBIDDEN.

You must use at least two of the following operators:

- `<<`
- `>>`
- `&`
- `\|`
- `^`
- `\~`

Here is a sample `main` function:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

int prettyprint_message(uint32_t, const char *);

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. */

prettyprint_message(0x02020001, NULL);
prettyprint_message(0x01040002, NULL);
prettyprint_message(0x01020002, "\"KREOG !!!\"");
prettyprint_message(0x01040001, "Brainfuck");
prettyprint_message(0x04040001, "\"Dark Moon of the side...\"");

return 0;
}
```

*main.c*

```
Terminal
~/B-PAV-242> gcc -Wall -Wextra -Werror -std=gnu99 main.c koalatchi.c && ./a.out |
cat -e
Invalid message.$
Request Nutrition Hungry "Kreog !"$
Notification Nutrition Eat Eucalyptus$
Notification Nutrition Defecate "CACA !"$
Notification Nutrition Eat Keytronic$
Error Nutrition Indigestion$
Request Entertainment NeedAffection$
Notification Education BeAwesome$
Notification Entertainment Bite "KREOG !!!"$
Notification Education TeachCoding Brainfuck$
Error Education BrainDamage "Dark Moon of the side..."$
```



You must NOT provide a main function.



## EXERCISE 8 - LOG

	Exercise: 08	points : 3
Log		
Turn-in directory: cpp_d01/ex08		
Compiler: gcc	Compilation flags: -Wall -Wextra -Werror -std=gnu99	
Makefile: No	Rules: n/a	
Files to turn in: log.h, log.c		
Notes: None		
Forbidden functions: None		

Logs play a very important role in computer science. Thanks to them, one can keep a record of everything that happened during a program execution. Taking a peek in `/var/log` shows the amount of information that are being kept, either by various applications or by the operating system itself.

We are going to create a few logging functions. They should provide a way to choose where the message we want to log will be written. It must be possible to print them to the standard output, the error output or even a file picked by the user. The default choice should be the error output. If a program logs a message to a file, it should be appended to the end of the file.

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

- Error
- Warning
- Notice
- Info
- Debug

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

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 set the maximum desired level to `Error`, thus only logging `Error`-level messages.

Messages will all be formatted as follows:

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

The date should be formatted as that returned by `ctime(3)`. You must obtain the system time with a call to `time(2)`.

In a `log.h` file, define a `LogLevel` enum containing all the enumerators mentioned above.

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



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

The `get_log_level` function returns the current log level.

The `set_log_level` function defines the log level to be used. This level is provided as a 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 `set_log_file` function lets users provide the name of the file to which messages should be logged. The filename is provided as a parameter. If another file was previously opened, it must be closed beforehand. The function returns 0 upon success and 1 otherwise.

The `close_log_file` function closes the current log file (if one is open) and resets the log output to the error output. If no file was open, this function returns without doing anything. It returns 0 if no error was encountered and 1 otherwise.

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

The `log_to_stderr` function sets the log output to the error output. If a file was previously opened, it must be closed beforehand. The function returns 0 if no error was encountered and 1 otherwise.

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

The messages' destination and the current log level must be held in global variables that **MUST NOT** be accessible through functions that were not defined in the `log.c` compilation unit.

Recommended reading: `fopen(3)`, `fprintf(3)`, `vfprintf(3)`, `ctime(3)`, `time(2)`, `stdarg(3)`.

Here is a sample `main` function:

```
#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_debugn");
    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;
}
```

*main.c*

```
Terminal
~/B-PAV-242> ls
log.c log.h main.c
~/B-PAV-242> gcc -Wall -Wextra -Werror -std=c99 main.c log.c
~/B-PAV-242> ./a.out
~/B-PAV-242> ls
a.out log.c log.h main.c out.log
~/B-PAV-242> 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 !
~/B-PAV-242> ./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.