B3 - C++ Pool

B-CPP-300

Day 01

C, Life, the Universe and everything else

2.2

Day 01

language: C

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.
- Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

All your exercises will be compiled with the `-std=gnu11 -Wall -Wextra` **flags**, unless specified otherwise.

Every function implemented in a header or any 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.

For each exercise, the files must be turned-in in a separate directory called **exXX** where XX is the exercise number (for instance `ex01`), unless specified otherwise.

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 <criterion/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 - Z IS FOR ZORGLUB

Turn in: `Makefile`, `z.c`
Compilation: using your `Makefile`
Executable name: `z`

Our Lord and Genius Zorglub, master of Zorgland, wants you to design the software for his new brain-washing machine to zorgmanize his enemies into zorgmen.
The machine works by sending a specific character into the mind of its victim.
The character must be chosen carefully using an algorithm devised by our Lord and Genius Zorglub.
The algorithm uses the prisoner ID to determine which character should be used.
An ID is a hexadecimal number that fits in a `uint64_t`, given as parameter to your program.
You must then display the character to be used, following the rules defined by the Grand Z:

- If the binary notation of the ID is a palindrome, it means that the victim is just a dummy used for test purpose. You must display the default brain-washing byte `0172` (octal) and validation byte `012` (octal).
- If the ID is a multiple of 13 and 29 and 89 or 41 and 71 or 67 or 7 and 43 and 47 and 53, you must display the string `"z\n"` and immediately stop your program.
- If the ID is `0x12345678`, `0x87654321`, `0x01111010` or `0x01011010`, you must display the following bits : `0111101000001010`.
- If one of the byte of the ID is worth `0x42`, it means that the victim is tough and the strongest character must be used : `'z'`. The byte `0x0A` is then displayed.
- If the last byte of the ID is null, the victim is just a minion of our enemies. As we don't care about them, we will brain-wash them using the last letter of the alphabet in lowercase. The output will then be flushed with a line feed.
- If the ID is a multiple of the sum of its 8 bytes, you must display the sixth character of the ASCII table starting from the end, followed by a `'\n'`.
- If the 8 bytes of the ID are identical, it means that the prisoner is one of our special guest. You will use the character `'Z'` in lowercase, and your program will end with a new line. This case override all the other cases.
- If the parameter contains non-hexadecimal characters, this is an error. The program must handle this case by displaying the error character `'z'` followed by a line feed.
- If the parameter does not fit in a `uint64_t`, this is an error. In this very specific case, the very specific character `'z'` must be used to brain-wash the victim, followed by the tenth character of the ASCII table.
- If there is no argument, the ID to be used is the return value of the function `time(NULL)`.
- If there is too many argument, your program must use the first valid argument starting from the end. If no argument is valid, we consider there is no argument and use the preceding rule.

Your program must *always* return `0` as it will never fail, even in case of invalid input.

Terminal

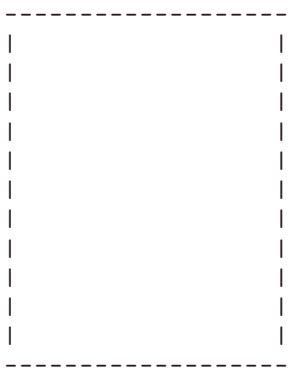
```
~/B-CPP-300> ./z "0x42242112" | cat -e
z$
~/B-CPP-300> ./z "invalid_ID" ; echo $?
z
0
```

Exercise 1 - The Menger Sponge

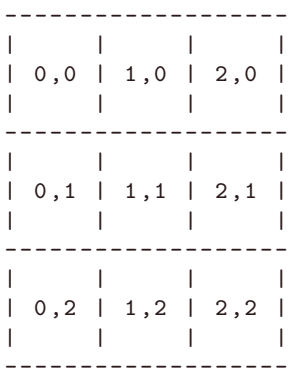
Turn in: `Makefile`, `menger.c`, `menger.h`, `main.c`
Compilation: using your `Makefile`
Executable name: `menger`

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 being "empty".
This process is then repeated for the 8 other squares.

Consider the following square:



Once processed, that square becomes:



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.

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

```
Level 0                 Level 1                 Level 2
--------------------    --------------------    --------------------
|# # #|# # #|# # #|      |# # #|# # #|# # #|    |# # #|# # #|# # #|
|# # #|# # #|# # #|      |# # #|# # #|# # #|    |#   #|#   #|#   #|
|# # #|# # #|# # #|      |# # #|# # #|# # #|    |# # #|# # #|# # #|
--------------------    --------------------    --------------------
|# # #|# # #|# # #|      |# # #|     |# # #|    |# # #|     |# # #|
|# # #|# # #|# # #|      |# # #|     |# # #|    |#   #|     |#   #|
|# # #|# # #|# # #|      |# # #|     |# # #|    |# # #|     |# # #|
--------------------    --------------------    --------------------
|# # #|# # #|# # #|      |# # #|# # #|# # #|    |# # #|# # #|# # #|
|# # #|# # #|# # #|      |# # #|# # #|# # #|    |#   #|#   #|#   #|
|# # #|# # #|# # #|      |# # #|# # #|# # #|    |# # #|# # #|# # #|
--------------------    --------------------    --------------------
```

Write a **program** that takes two arguments:

1. the size of the original square (always a positive number multiple of 3^{level}),
2. the number of levels.

It must display:

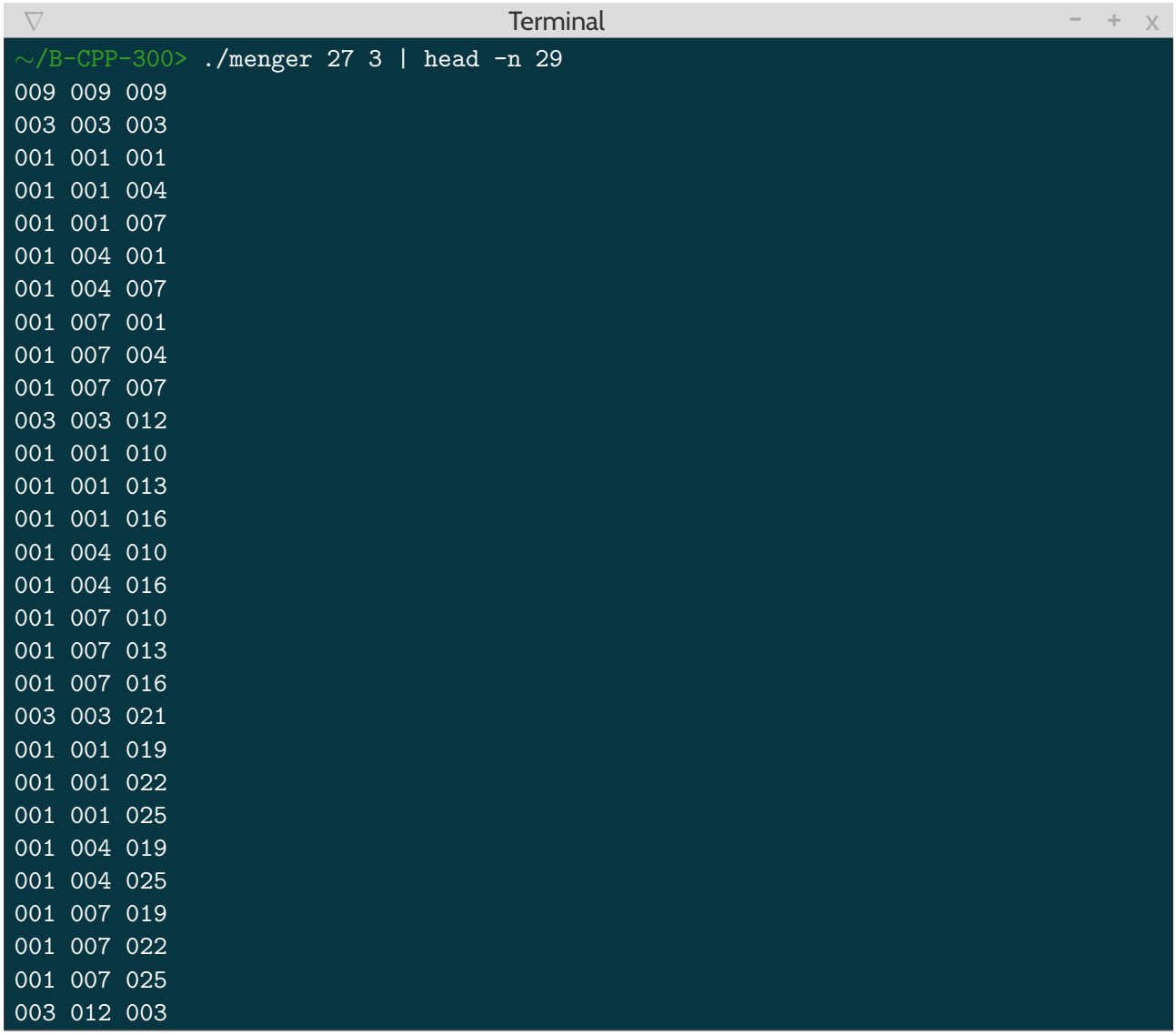
1. the size, and the abscissa and ordinate of the top left corner of the main empty square,
2. recursively for each sub-square, the size, the abscissa and the ordinate of their empty square.

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

For example:

Terminal

```
~/B-CPP-300> ls
Makefile main.c menger menger.c menger.h
~/B-CPP-300> ./menger 3 1
001 001 001
~/B-CPP-300> ./menger 9 1
003 003 003
~/B-CPP-300> ./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-CPP-300> ./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

Turn in: `bitmap.h`, `bitmap_header.c`

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 adress 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 `bmp_header_t` and `bmp_info_header_t` structures, respectively representing the file header and the image header.

The `bmp_header_t` structure must have the following fields:

```
magic
size
_app1
_app2
offset
```

The `bmp_info_header_t` structure must have the following fields:

`size`	`raw_data_size`
`width`	`h_resolution`
`height`	`v_resolution`
`planes`	`palette_size`
`bpp`	`important_colors`
`compression`	

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

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

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

The `make_bmp_header` function has the following signature:

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

- `header`: a pointer to the `bmp_header_t` 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(bmp_info_header_t *header, size_t size);
```

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

About the pictures you'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 `bmp_header_t` should have their order reversed.
Indeed, on a little endian computer, any number's bytes are reversed in terms of memory representation.

PADDING

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

Terminal

```
~/B-CPP-300> 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;
}
~/B-CPP-300> 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.

```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;
}
```

Example

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"

// not checking your return values is naugthy naughty naughty

void    write_bmp_header(int fd)
{
    bmp_header_t header;

    make_bmp_header(&header, 32);
    write(fd, &header, sizeof(header));
}

void    write_bmp_info_header(int fd)
{
    bmp_info_header_t info;

    make_bmp_info_header(&info, 32);
    write(fd, &info, sizeof(info));
}

void    write_32bits(int fd, uint32_t pixel)
{
    for (size_t i = 0; i < 32 * 32; ++i)
        write(fd, &pixel, sizeof(pixel));
}

int     main(void)
{
    int fd = open("32px.bmp", O_CREAT | O_TRUNC | O_WRONLY, 0644);

    write_bmp_header(fd);
    write_bmp_info_header(fd);
    write_32bits(fd, 0x00FFFFFF);
    close(fd);
    return EXIT_SUCCESS;
}
```

Terminal

```
~/B-CPP-300> 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 |................|
```

Exercise 3 - It must be nice from up there

Turn in: `pyramid.c`

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 = 12
```

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

```
int pyramid_path(int size, const 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 distances between rooms

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 interesting 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 4 - Draw me a square

Turn in: `drawing.h`, `drawing.c`, `bitmap.h`, `bitmap_header.c`

It is now time to fill in the pictures you created in Exercise 2.

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 must 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 color examples:

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

In a `drawing.h` file, create a `point_t` 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 must draw a square of a given size to a given position.

```
void draw_square(uint32_t **img, const point_t *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 the square's sides
- `color`: the color of the square to be drawn

It must 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"

// not checking your return values is naugthy naughty naughty

void    write_bmp_header(int fd);

void    write_bmp_info_header(int fd);

void    initialize_image(size_t size, unsigned int *buffer, unsigned int **img)
{
    memset(buffer, 0, size * size * sizeof(*buffer));
    for (size_t i = 0; i < size; ++i)
        img[i] = buffer + i * size;
}

void    create_image(size_t size, unsigned int *buffer, unsigned int **img)
{
    point_t p = { 0, 0 };

    initialize_image(size, buffer, img);
    draw_square(img, &p, size, 0x0000FFFF);
    p.y = 10;
    draw_square(img, &p, 22, 0x00CC0000);
}

void    create_bitmap_from_buffer(size_t size, unsigned int *buffer)
{
    int fd = open("square.bmp", O_CREAT | O_TRUNC | O_WRONLY, 0644);

    write_bmp_header(fd);
    write_bmp_info_header(fd);
    write(fd, buffer, size * size * sizeof(*buffer));
    close(fd);
}

int     main(void)
{
    size_t size = 64;
    unsigned int *buffer = malloc(size * size * sizeof(*buffer));
    unsigned int **img = malloc(size * sizeof(*img));

    create_image(size, buffer, img);
    create_bitmap_from_buffer(size, buffer);
    return EXIT_SUCCESS;
}
```

EXERCISE 5 - FOOK THIS, SERIOUSLY

Turn in: `ex05.h`

Create the `ex05.h` file needed for the following code to compile and generate the expected output.

```c
#include <stdlib.h>
#include <stdio.h>
#include "ex05.h"

int main(void)
{
    foo_t 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;
}
```

```
Terminal
~/B-CPP-300> ls
ex05.h main.c
~/B-CPP-300> gcc -Wall -Wextra -std=gnu11 main.c
~/B-CPP-300> ./a.out
1
1
1
1
0000CAFE
```

Exercise 6 - Draw me a sponge

Turn in: `Makefile`, `menger.c`, `menger.h`, `main.c`, `drawing.h`, `drawing.c`, `bitmap.h`, `bitmap_header.c`
Compilation: using your `Makefile`
Binary name: `menger_face`

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.

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 must take as arguments

- the name of the image file to create,
- the size of the image's sides,
- the depth for the Menger's Sponge.

If the number of arguments is incorrect, 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 must be colored in black, and empty squares in grey.
The colors must actually be tightly coupled to the current depth of the Sponge.
Each component of the color must be equal to 0xFF divided by the remaining depth level plus one.
Thus, the smallest empty squares of the Sponge must 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	Value	Color
1	255 / 3	0x00555555
2	255 / 2	0x007F7F7F
3	255 / 1	0x00FFFFFF

Exercise 7 - Koalatchi

Turn in: `koalatchi.c`

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** or **Education**.

The message header has the following structure:

1. The **message type** on 1 byte.
   Possible values are 1 (*Notification*), 2 (*Request*) or 4 (*Error*).
2. The **application domain** on 1 byte.
   Possible values are 1 (*Nutrition*), 2 (*Entertainment*) or 4 (*Education*).
3. A unique value describing **the message**, on 2 bytes.

Here are the 16 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 teachs 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.
- `return`: if the message is valid, the function returns 0. Otherwise it returns 1.

This function must 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 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:

```c
#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, "\"POOH!\"");
    prettyprint_message(0x01010001, "Keytronic");
    prettyprint_message(0x04010001, NULL);
    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;
}
```

Terminal

```
~/B-CPP-300> gcc -Wall -Wextra -Werror main.c koalatchi.c && ./a.out | cat -e
Invalid message.$
Request Nutrition Hungry "Kreog!"$
Notification Nutrition Eat Eucalyptus$
Notification Nutrition Defecate "POOH!"$
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..."$
```

EXERCISE 8 - LOG

Turn in: `log.h`, `log.c`, `log_config.c`

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.

You are going to create a **few logging functions**.
They should provide a way to choose where the message we want to log must 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 must be associated to each message.
These levels are inspired by `syslog(3)`:

- `ERROR`
- `WARNING`
- `NOTICE`
- `INFO`
- `DEBUG`

All these levels must be defined in a `log_level` 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 looged must be `ERROR` and `WARNING`-level messages.

The default behavior must set the maximum desired level to `ERROR`, thus only logging `ERROR`-level messages.

Messages must 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 `log_level` enum containing all the enumerators mentioned above.
In the file `log_config.c`, implement the following functions:

```
enum log_level get_log_level(void);
enum log_level set_log_level(enum log_level);
int set_log_file(const char *);
int close_log_file(void);
```

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

```
int log_to_stdout(void);
int log_to_stderr(void);
void log_msg(enum log_level, 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.

As global variables are forbidden, the messages' destination and the current log level must be held as static variables inside wrapper functions of your own design.

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

Here is a sample `main` function:

```c
#include <stdio.h>
#include <stdlib.h>
#include "log.h"

void    test_debug(void)
{
    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");
}

void    test_warning(void)
{
    set_log_level(WARNING);
    log_msg(INFO, "This is info\n");
    log_msg(ERROR, "KREOG!\n");
}

int     main(void)
{
    set_log_file("out.log");
    test_debug();
    test_warning();
    close_log_file();
    return EXIT_SUCCESS;
}
```

Terminal

```
~/B-CPP-300> ls
log.c log.h main.c
~/B-CPP-300> gcc -Wall -Wextra -std=gnu11 main.c log.c
~/B-CPP-300> ./a.out
~/B-CPP-300> ls
a.out log.c log.h main.c out.log
~/B-CPP-300> 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-CPP-300> ./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!
```