{EPITECH}

C CODING STYLE_

< KEEP YOUR CODE NICE AND CLEAN />

C CODING STYLE

The *Epitech C Coding Style* is a set of rules that have been created within the school, and that you have to follow.

It covers various aspects of programming in C, from the overall organization of the repository to the individual lines of code.

It is compulsory on all programs written in C as part of Epitech's projects, **regardless** of the year or unit, as long as a language to program in is imposed.

It applies to **all source** (`.c`) and **header files** (`.h`) present in the repository, as well as **Makefiles**.

Adopting a coding style makes reading code written by others easier. As such, it facilitates group work, as well as help given to you by the educational team and the assistants.

It is also an excellent way to encourage structuring the code and making it clearer, and thus facilitates:

- ✓ its reading;
- ✓ its debugging;
- ✓ its maintenance;
- ✓ its internal logic definition;
- ✓ its reusability;
- ✓ writing tests;
- ✓ adding new features;
- ✓ and even more...

A clean and structured code always feels nice to look at, so give yourself this treat. ;)

When you are facing a choice and you do not know what decision to make, always ask yourself which one helps you make your code clearer, ergonomic and flexible.

In case of uncertainty or ambiguity regarding the principles and rules specified in this document, please refer to your local educational manager.

Any attempt to bypass the coding style rules will result in an entire project invalidation.

Rules are categorized into 4 severity levels: **fatal** ⛔, **major** ⬆, **minor** ⬇ and **info** ℹ.

Fatal rules ⛔ are related to the **objective itself of programming in C**. Infringing **even once** a fatal rule will make your project rejected and **not evaluated at all**.

Major rules ⬆ are related to the **structure of the code** and to practices that are detrimental to the production of a code of good quality. Infringing **any** of the major rules (even once) is a **major problem** and must be corrected as a priority concern.

Minor rules ⬇ are generally related to the **visual presentation of the code**, which can make the code difficult to read if not followed consistently. Repeatedly infringing minor rules must be avoided, as it creates inconsistently formatted code, which in turn makes it harder to read.

Info rules ℹ are related to **specific trivial points** that are not as important as other rules. Each of these rules are however anchored in good practices, and should as such be followed to ensure a code of the best quality possible.

There are many ways to produce unclean code, and as such many rules to follow in order to avoid them. Even though one cannot mention all of them in this document, they still have to be respected.
We call them *implicit rules* when not explicitly defined in this document.

Implicit rules are considered as info ℹ.

The *Coding Style* is a purely syntactic convention, so it can not be used as an excuse if your program does not work. ;)

Although following the coding style is not required in all projects, this is not a reason for not always sequencing and structuring your code.
Most of the rules in this coding style can be applied to all languages, so they can be useful when you are doing projects in different languages.
It is easier and quicker to follow the coding style from the beginning of a project rather than to adapt existing code at the end.

This document is inspired by the Linux Kernel Coding Style, and is freely adapted from Robert C. Martin's excellent book *Clean Code*.

Banana

The adherence to the coding style is **partially** checked during evaluations by a tool called the *Bot Analyzing Nomenclature And Nonsensical Arrangements*, better known as *Banana*.

You can (and should) also use this tool to check that your code follows a good portion of the rules.
Other rules are checked manually, with the great tool that are your eyes.

The rules are tagged with three possible levels of support by Banana:

- ✓  → the rule is completely checked by Banana;
- ✓  → the rule has to be checked manually (Banana does not support it);
- ✓ → some parts of the rule are checked by Banana, other parts have to be checked manually.

Using Banana

Using Banana is very simple, you just need to follow the instructions on its source repository.

The source repository is accessible to all Epitech students.

If you find any problem or have any question regarding Banana, you can open an issue there, and a Banana developer will happily answer you.

You can even contribute to Banana yourself if you want! ;)

C-O - Files organization

C-O1 - Contents of the repository

The repository **must not** contain **compiled** (`.o`, `.a`, `.so`, ...), **temporary** or **unnecessary** files (`*~`, `#*#`, etc.).

Git has a wonderful way to help you keep your repository clean. ;)

C-O2 - File extension

Sources in a C program must **only** have `.c` **or** `.h` extensions.

C-O3 - File coherence

A source file must match a **logical entity**, and group all the functions associated with that entity.

Grouping functions that are **not related** to each other in the same file has to be **avoided**.

You are allowed to have **10 functions** (including at most **5 non-static functions**) in total per file.

Beyond these amounts, you **must** subdivide your logical entity into several sub-entities.

C-O4 - Naming files and folders

The name of the file must define the logical entity it represents, and thus be **clear, precise, explicit and unambiguous**.

For example, files like `string.c` or `algo.c` are probably incorrectly named.
Names like `string_toolbox.c` or `pathfinding.c` would be more appropriate.

All file names and folders must **be in English, according to the `snake_case` convention** (that is, only composed of lowercase, numbers, and underscores).

Abbreviations are tolerated as a way to significantly reduce the size of a name **only** if it does not lose its meaning.

C-G - Global scope

Multiline statements

Multiline statements are allowed.

Here are examples of properly segmented multiline statements:

```
bool is_between(unsigned int n, unsigned int low_bound,
    unsigned int high_bound, const char *fail_message);

int main(void)
{
    printf("[%s] %s: %d\n", get_element_type(), get_element_name(),
        get_element_value());
    my_putstr("Writing multiline statements in C is easy,"
        " you just need to break the line, and you are done!");
    if (call_to_a_function_with_a_long_but_descriptive_name()
        && (call_to_another_function_inside_brackets()
            || call_to_function_if_the_first_one_did_not_succeed())
        && final_call_to_function_to_demonstrate_multiline()) {
        i_am_a_teapot(418);
    }
}
```

Do not use the backslash character (\) to break lines in C files, because it will only *visually* break the line.
As such, you will be at risk of getting into trouble regarding the coding style!

Language extensions

Language extensions are **not supported**.

Using them might lead to an undefined behaviour by Banana and, as such, to problems in your evaluation.

Some examples are:

OK	Not supported
`[[noreturn]]`	`__attribute__((noreturn))`
`#ifndef/#define/#endif`	`#pragma once`
`\x1B`	`\e`

C-G1 - File header

C files (`.c`, `.h`, ...) and every Makefiles must always start with the **standard header** of the school.
This header is created in Emacs using the `Ctrl + c` and `Ctrl + h` command.

For C files:

```
/*
** EPITECH PROJECT, [YEAR]
** [NAME_OF_THE_PROJECT]
** File description:
** No file there, just an epitech header example.
** You can even have multiple lines if you want!
*/
```

For Makefiles:

```
##
## EPITECH PROJECT, [YEAR]
## [NAME_OF_THE_PROJECT]
## File description:
## No file there, just an epitech header example.
## You can even have multiple lines if you want!
##
```

Always add a meaningful description of the file, you have a unlimited amount of lines to do so.

C-G2 - Separation of functions

Inside a source file, implementations of functions must be separated by **one and only one empty line**.

C-G3 - Indentation of preprocessor directives

The preprocessor directives must be **indented according to the level of indirection**.

Indentation must be done in the same way as in the C-L2 rule (groups of 4 spaces, no tabulations). **However**, preprocessor directives must be **indented independently of all the other code**.

```
#ifndef WIN32
    #include <stdbool.h>
    #if defined(__i386__) || defined(__x86_64__)
const size_t PAGE_SIZE = 4096;
    #else
        #error "Unknown architecture"
    #endif

struct coords {
    int x;
    int y;
};
#endif
```

C-G4 - Global variables

Global variables must be **avoided** as much as possible.

Only global **constants** should be used.

A constant is considered as such if and only if it is correctly marked with the `const` keyword. Watch out, this keyword follows some particular and sometimes surprising rules!

```
const float GOLDEN_RATIO = 1.61803398875;               /* OK */
int uptime = 0;                                         /* C-G4 infraction */
```

Static variables are also global variables.

C-G5 - `include`

`include` directives must **only** include C header (`.h`) files.

C-G6 - Line endings

Line endings must be **done in UNIX style** (with `\n`), and must **never end with a backslash (\)**.

`\r` must not be used at all, anywhere in the files.

`git config` can help you keep your lines correctly ended.

Have a look at the *Multiline statements* section up above to see how to properly break lines.

C-G7 - Trailing spaces

No trailing spaces must be present at the end of a line.

C-G8 - Leading/trailing lines

No leading empty lines must be present.

No more than 1 trailing empty line must be present.

Make sure you also follow the C-A3 rule.

C-G9 - Constant values

Non-trivial constant values should be defined either as a global constant or as a macro.

This greatly helps you when you want to modify an important value in your program, because you do not need to find all occurences of this value scattered throughout your code, and only need to change it in one place.

C-G10 - Inline assembly

Inline assembly must **never** be used.

Programming in C must be done... in C.

C-F - Functions

C-F1 - Coherence of functions

A function should only do **one thing**, not mix different levels of abstraction, and respect the single-responsibility principle (a function should be changed only for one reason).

For example, a call to `malloc()`, a call to `allocate_user()`, and a call to `create_user()` all have 3 different levels of abstraction.

C-F2 - Naming functions

The name of a function must **define the task it executes** and must **contain a verb**.

For example, the `vowels_nb()` and `dijkstra()` functions are incorrectly named.
`get_vowels_number()` and `search_shortest_path()` are more meaningful and precise.

All function names must **be in English, according to the `snake_case` convention** (meaning that it is composed only of lowercase, numbers, and underscores).

Abbreviations are tolerated if they significantly reduce the name without losing meaning.

C-F3 - Number of columns

The length of a line must not exceed **80 columns** (not to be confused with 80 *characters*).

A tab represents 1 character, but several columns.

Even though this rule especially applies to functions, **it applies to all C files, as well as Makefiles**.

C-F4 - Number of lines

The body of a function should be as **short as possible**, and must not exceed **20 lines**.

```c
int main(void)          /* this function is 2-line-long */
{
    printf("hello, world\n");
    return 0;
}
```

> The maximum length of a function is inversely proportional to the complexity and indentation level of that function.
> So, if you have a conceptually simple function that is just one long (but simple) case-statement, where you have to do lots of small things for a lot of different cases, it's OK to have a longer function.
>
> **— Linus Torvalds, Linux Kernel Coding Style —**

C-F5 - Number of parameters

A function must not have more than **4 parameters**.
Writing variadic functions is allowed, but they must not be used to circumvent the limit of 4 parameters.

C-F6 - Functions without parameters

A function (or function pointer) taking no parameters **must take `void` as a parameter** in the function declaration.

```
phys_addr_t alloc_frame();                  /* C-F6 infraction */
phys_addr_t alloc_frame(void);              /* OK */
```

The two syntaxes above have different meanings, and have different interesting behaviours.

C-F7 - Structures as parameters

Structures must be passed to functions using a **pointer**, **not by copy**.

```
void make_some_coffee(struct my_struct *board, int i);  /* OK */

void make_some_poison(struct my_struct board, int i);   /* C-F7 infraction */
```

C-F8 - Comments inside a function

There **must be no comment** within a function.
The function should be readable and self-explanatory, without further need for explanations inside it.

The length of a function is inversely proportional to its complexity, so a complicated function should be short.
In that case, a header comment (just above the prototype) should be enough to explain it.

C-L - Layout inside a function scope

C-L1 - Code line content

A line must correspond to **only one statement**.

We roughly define a statement as a being a short piece of code that either:

- ✓ creates/sets a variable/constant;
- ✓ calls a function without using its return value;
- ✓ checks a condition as part of an `if`, `else if`, `switch`, `while`, `do while`, or `for`;
- ✓ makes the function `return`.

Typical situations to avoid include:

- ✓ several assignments on the same line;
- ✓ several semicolons on the same line used to separate several statements;
- ✓ an assignment in a control structure expression;
- ✓ a condition and a statement on the same line.

The only exception to this rule is the `for` loop control structure, for which **one** statement is allowed in each of the three parts (initialization, loop condition, and post-iteration operation).

OK	C-L1 infraction
`a = 0;`	`a = b = c = 0;`
`a++;`	`a++; b++;`
`if (ptr != NULL)`	`if ((ptr = malloc(42))!= NULL)`
	`if (cond)return (ptr);`
`for (int i = 0; i < 42; i++)`	`for (int i = j = 0; i < 42; i++)`
	`for (int i = 0; i < 42; i++, j--)`
`str[i] = 'A';`	`str[i++] = 'A';`
`return my_strlen(name);`	

Ways to circumvent this rule are **not** allowed:

```
((a != c) || (b != c)) && printf("Hello, world!\n");
```

is forbidden as it just a way to sneakily blend two statements into one:

```
if ((a != c) || (b != c)) {
    printf("Hello, world!\n");
}
```

C-L2 - Indentation

Each indentation level must be done by using **4 spaces**.
No tabulations may be used for indentation.

When entering a new scope (e.g.: control structure or function call argument list), the indentation level must be incremented.

```
// OK
int main(void)
{
    char letter = 'H';
    int number = 14;

    if (letter == 'H') {
        my_putchar('U');
    } else if (letter == 'G') {
        if (number != 10)
           my_putchar('O');
        else {
            my_putnbr(97);
        }
    }
    while (my_function_with_a_long_name( // This adds two indentation levels
            letter, number)) {
        my_putchar('A');
    }
}

// Incorrect
int main(void)
{
int i;

    if (true) {
    return (0);
    }
}
```

Block comments' bodies can be freely indented (with spaces):

```
// The block comment below is valid, even though it is indented with only one space
/**
 * @brief Something
 *
 * @param path
 * @return void*
 */
void *something(const char *path);
```

C-L3 - Spaces

When using a space as a separator, **one and only one space** character must be used.

Tabulations **cannot** be used as a separator.

Always place a **space after a comma or a keyword** (**if it has arguments**).

However, there must be **no spaces** between the name of a function and the opening bracket, after a unary operator, before a semicolon, or before a comma.

In the precise case of a `for` control structure, if a semicolon inside the brackets is not **immediately** followed by another semicolon, it **must** be followed by a space.

All binary and ternary operators must be separated from their arguments by a **space on both sides**.

`return` is a *keyword*, but `sizeof` is an *unary operator*.
`else` keyword must be surrounded by a space.

OK	C-L3 infraction
`return 1;` as well as `return (1)`	`return(1);`
`break;`	`break ;`
`add_numbers(1, 2);`	`add_numbers(1 , 2);`
`sum = term1 + 2 * term2;`	`sum = term1+2*term2;`
`s = sizeof(struct file);`	`s = sizeof (struct file);`
`for (size_t i; str[i] != '\0'; i++)`	`for (size_t i;str[i] != '\0'; i++)`

C-L4 - Curly brackets

Opening curly brackets must be **at the end of the line**, after the content it precedes, except for functions definitions where they must be placed alone on their line.
Closing curly brackets must be **alone on their line**, except in the case of `else`/`else if`/`do while` control structures, `enum` declarations, or structure declarations (with or without an associated `typedef`).

In the case of a single-line scope, omitting curly brackets is tolerated, but you should think about all the modifications you will have to make if you want to add a new statement to the block. This can also introduce some nasty bugs!

```
if (cond) {return ptr;}          /* C-L1 & C-L4 infractions */
while (cond) {                   /* OK */
    do_something();
}
if (cond)
{                                /* C-L4 infraction */
    ...
} else {                         /* OK */
    ...
}
if (cond) {                      /* OK */
    ...
}                                /* C-L4 infraction */
else {
    ...
}
if (cond)                        /* Tolerated */
    return ptr;
int print_env(void)              /* OK */
{
    ...
}
int print_env(void) {            /* C-L4 infraction */
    ...
}
struct dummy {                  /* OK */
    int var;
};
struct dummy
{                                /* C-L4 infraction */
    int var;
};
```

Even though this primarily applies to the contents of functions, **this rule also applies to code outside functions**, including header files'.

C-L5 - Variable declarations

Variables must be declared **at the beginning of the function**.
Only one variable must be declared per statement.

The `for` control structures may also optionally declare a variable in their initialization part.

Nothing prevents you from declaring and assigning a variable on the same line.

```
long calculate_gcd(long a, long b)
{
    long biggest, smallest;            /* C-L5 infraction */

    biggest = MAX(a, b);
    smallest = MIN(a, b);
    long rest;                         /* C-L5 infraction */
    while (smallest > 0) {
        rest = biggest % smallest;
        biggest = smallest;
        smallest = rest;
    }
    return a;
}

int main(void)
{
    int forty_two = 42;                         /* OK */
    int max = 12;                               /* OK */

    for (unsigned int i = 0; i < max; i++) {    /* OK */
        calculate_gcd(forty_two, max);
    }
    return 0;
}
```

C-L6 - Blank lines

A blank line must **separate the variable declarations from the remainder** of the function.
No other blank lines must be present in the function.

```
int sys_open(char const *path)
{
    int fd = thread_reserve_fd();
    struct filehandler *fhandler = NULL;
                                                /* OK */

    if (fd < 0) {
        return -1;
    }
    if (fs_open(path, &fhandler)) {
        thread_free_fd(fd);
        return -1;
    }
                                                /* C-L6 infraction */
    thread_set_fd_handler(fd, fhandler);
    return fd;
}
```

No blank line is necessary if there are no variable declarations in the function.

C-V - Variables and types

C-V1 - Naming identifiers

All identifier names must **be in English, according to the `snake_case` convention** (meaning it is composed exclusively of lowercase, numbers, and underscores).

The type names defined with `typedef` must **end with `_t`**.
The names of **macros and global constants** and the content of **enums** must be written in `UPPER_SNAKE_CASE`.

```
#define IS_PAGE_ALIGNED(x) (!((x) & (PAGE_SIZE - 1)))     /* OK */
enum arch {                                               /* OK */
    I386 = 0,
    X86_64,
    ARM,
    ARM64,
    SPARC,
    POWERPC,
};
const float PI = 3.14159;                                  /* OK */
typedef int age;                                           /* C-V1 infraction */
typedef struct int_couple pixel_t;                         /* OK */
```

Abbreviations are tolerated as long as they significantly reduce the name length without losing meaning.

C-V2 - Structures

Variables can be grouped together into a structure if and only if they form **a coherent entity**.
Structures must be kept **as small as possible**.

```
struct person {                          /* OK */
    char *name;
    unsigned int age;
    float salary;
};

struct data {                           /* C-V2 infraction */
    struct person player;
    unsigned int width;
    unsigned int length;
    unsigned int score;
    int i;
};
```

C-V3 - Pointers

The asterisk (*) must be **attached to the associated identifier on its right**, with no spaces in between.

It must also be **preceded by a space**, except when it is itself preceded by an opening parenthesis or another asterisk.

This includes using the asterisk to declare or dereference a pointer.

When used in a cast, the asterisk must have a space on its left side, but not on its right side.

OK	C-V3 infraction
`int *a;`	`int* a;`
`char **argv;`	`char**argv;`
`char *const *array;`	`char * const *array;`
`int a = 3 * b;`	`int a = 3*b;`
`int strlen(char const *str);`	`int strlen(char const* str);`
`char *nbr_to_str(int i);`	`char* nbr_to_str(int i);`
`my_put_nbr(*ptr);`	`my_put_nbr(* ptr);`
`(int *)ptr;`	`(int*)ptr;` as well as `(int * )ptr;`
`void (*func_ptr)(int)= &func;`	`void (* func_ptr)(int)= &func;`

This rule only applies in pointer context.

C-C - Control structures

Unless otherwise specified, all control structures are allowed.

C-C1 - Conditional branching

Nested conditional branches with a depth of 3 or more must be avoided.

If you need multiple levels of branches, you probably need to refactor your function into sub-functions.

```
if (...) {                          /* OK */
    do_something();
} else if (...) {
    do_something_else();
} else {
    do_something_more();
}

if (...) {
    do_something();
} else if (...) {
    do_something_else();
} else if (...) {
    do_something_more();
} else {                            /* C-C1 infraction */
    do_one_last_thing();
}

while (...) {                       /* OK */
    if (...) {
        do_something();
    }
}

while (...) {                       /* C-C1 infraction */
    for (...) {
        if (...) {
            do_something()
        }
    }
}
```

`else if` branching does not add one, but **two** levels of depth, as it is considered to be an `if` inside an `else`.

```
// This code...
if (...) {
    do_something();
} else if (...) {
    do_something_else();
} else if (...) {                   /* C-C1 infraction */
    do_one_last_thing();
}

// ... is unfolded as this
if (...) {
    do_something();
} else {
    if (...) {
        do_something_else();
    } else {
        if (...) {                  /* C-C1 infraction */
            do_one_last_thing();
        }
    }
}
```

```
// This code...
if (...) {
    do_something();
} else if (...) {
    while (...) {                   /* C-C1 infraction */
        do_something_else();
    }
}

// ... is unfolded as this
if (...) {
    do_something();
} else {
    if (...) {
        while (...) {               /* C-C1 infraction */
            do_something_else();
        }
    }
}
```

Arrays of function pointers and `switch` instructions are very useful when you want to have numerous different behaviours that can result from the check of an element.
Make sure to choose the most suitable one.

C-C2 - Ternary operators

The use of ternary operators is **allowed as far as it is kept simple and readable**, and if it does not obfuscate code.

You must never use **nested or chained ternary operators**.
You must **always use the value** produced by a ternary operator (by assigning it to a variable or returning it for example).

```
parity_t year_parity = (year % 2 == 0) ? EVEN : ODD;              /* OK */
return (a > 0 ? a : 0);                                           /* OK */
unsigned int safe_sum = is_sum_overflow(a, b) ? 0 : a + b;        /* OK */
char *result = is_correctly_formatted(str) ? str : format(str); /* OK */
int a = b > 10 ? c < 20 ? 50 : 80 : e == 2 ? 4 : 8;               /* C-C2 infraction */
already_checked ? go_there() : check();                           /* C-C2 infraction */
first() ? second() : 0;                                           /* C-C2 infraction */
```

C-C3 - goto

Using the `goto` **keyword is forbidden**, because it can very quickly participate in the creation of infamous spaghetti code, which is completely unreadable.

C-H - Header files

C-H1 - Content

Header files must only contain:

- ✓ **function prototypes**,
- ✓ **type declarations**,
- ✓ **structure declarations**,
- ✓ **enumeration declarations**,
- ✓ **global variable/constant declarations**,
- ✓ **macros**,
- ✓ **static inline** functions.

All these elements must **only** be found in header files, and thus not in source files.

Including a header from another header is allowed as long as the header file itself needs it.
If a source file requires it, but not the header file itself, it should then be included in the source file instead.

In order to keep your code simple and readable, you should **not use conditional preprocessor directives in source files**.

C-H2 - Include guard

Headers must be protected from **double inclusion**, by using the `#ifndef`, `#define`, and `#endif` preprocessor directives.

C-H3 - Macros

Macros must match **only one statement**, and **fit on a single line**.

```c
#define PI                  3.14159265358979323846      /* OK */
#define DELTA(a, b, c)      ((b) * (b) - 4 * (a) * (c)) /* OK */
#define PRINT_NEXT(num)     {num++; printf("%d", num);} /* C-H3 infraction */
#define ERROR_MESSAGE       "Multiline macros" \        /* C-G6 infraction */
                            " have to be avoided"       /* C-H3 infraction */
```

Using a macro to shorten a long expression is **not** a valid reason to use a macro:

```c
// Unnecessary and obfuscates the code
#define WIN (data->object->scene->state->window)
```


C-A - Advanced

C-A1 - Constant pointers

When creating a pointer, if the pointed data is not (or should not be) modified by the function, it should be marked as **constant** (`const`).

C-A2 - Typing

Prefer the **most accurate types possible** according to the use of the data.

```
int counter;                                    /* C-A2 infraction */
unsigned int counter;                           /* OK */
unsigned int get_obj_size(void const *object)   /* C-A2 infraction */
size_t get_obj_size(void const *object)         /* OK */
```

Useful types include `size_t`, `ptrdiff_t`, `uint8_t`, `int32_t`, and more...

C-A3 - Line break at the end of file

Files must **end with a line break**.

Terminal

```
$> cat -e correct.c
int main(void) {$
    return 0;$
}$
$> cat -e incorrect.c
int main(void) {$
    return 0;$
}
```

The reason for this is tied to the POSIX's definition of a line:

> A sequence of zero or more non-<newline> characters plus a terminating <newline> character.
>
> **— The Open Group Base Specifications Issue 8, 2024 edition —**

C-A4 - static

Global variables and functions that are not used outside the compilation unit to which they belong should be **marked with the `static` keyword**.

Be careful not to confuse the different uses of the `static` keyword.
It does very different things depending on where you use it.

v 8