



Epitech Documentation

C Coding Style

Keep your code nice and clean





The *Epitech Coding Style* is a set of rules, guidelines and programming conventions that has been created within the school, and that you have to respect.

It applies to:

- the organization of the delivery folder,
- the denomination of the identifiers,
- the overall presentation (paragraphs),
- the local presentation (lines),
- source files and headers,
- Makefiles.



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

It is compulsory on all programs written in C as part of Epitech projects, regardless of the year or unit. It applies to all source files (.c) and headers (.h), as well as Makefiles.



Although 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* apply to all languages, so it can be useful when you're doing projects in different languages.



It's easier and quicker to follow the guide style from the beginning of a project rather than to adapt an existing code at the end.

The existence of this *Coding Style* is justified by the need to standardize the writing of programs within the school, in order to facilitate group work.

It is also an excellent way to encourage structuring and clarity of the code and thus facilitate :

- its reading,
- its debugging,
- its maintenance,
- its internal logic,
- its reuse,
- writing tests,
- adding new features ...



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



However, if you provide a **complete, relevant, accurate justification with a long-term view** (cleanliness, legibility, code flexibility, optimization, etc.), you can infringe some of the *Coding Style* points.



The relevance of this justification is left to the discretion of the proofreader, so it is preferable to present a strong argumentation or to abstain.

In case of uncertainty or ambiguity in the principles specified in this document refer to your local education manager.

There are 3 levels of severity: **major** 🚫, **minor** 🟢 and **info** ⓘ.

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



Implicit rules are considered as infos ⓘ.



This document is inspired by the [Linux Kernel Coding Style](#) , and is freely adapted from Robert C. Martin's excellent book `Clean Code`.



Some tools (such as [Editor Config](#)) might simplify the task.



O- FILES ORGANIZATION

+ O1- CONTENTS OF THE DELIVERY FOLDER

🚫 Your delivery folder should contain only **files required for compilation**.



This means no compiled (.o, .gch, .a, .so, ...), temporary or unnecessary files (*~* #, *.d, toto,...).

+ O2- FILE EXTENSION

✅ Sources in a C program should only have extensions **.c or .h**.

+ O3- FILE COHERENCE

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



Beyond 5 functions in your file, you should probably subdivide your logical entity into several sub-entities.

+ O4- NAMING FILES AND FOLDERS

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



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

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



Abbreviations are tolerated to the extent that they can significantly reduce the size of the name without losing the meaning.



G- GLOBAL SCOPE

+ G1- FILE HEADER

🔴 The source files (.c, .h, Makefile,...) should always start with the **standard header** of the school. This header is created in Emacs using the C-c C-h command.

```
/*
** EPITECH PROJECT, $YEAR
** $NAME_OF_THE_PROJECT
** File description:
**     No file there, just an epitech header example
**/
```

+ G2- SEPARATION OF FUNCTIONS

🟢 Inside a source file, **one and only one empty line** should separate the implementations of functions.

+ G3- INDENTATION OF PRE-PROCESSOR DIRECTIVES

🟢 The pre-processor directives should be **indented according to the level of indirection**.

```
#ifndef WIN32
    #if defined(__i386__) || defined(__x86_64__)
        size_t PAGE_SIZE = 4096;
    #else
        #error "Unknown architecture"
    #endif
#endif
```



The L2 rule also applies for pre-processor directives.



+ G4- GLOBAL VARIABLES

✔ Any **non-trivial constant** should be defined by a constant global variable.



Non-constant global variables should be **avoided** as much as possible.



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

```
const float GOLDEN_RATIO = 1.61803398875;           /* OK */
```

+ G5- 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.



F- FUNCTIONS

+ F1- COHERENCE OF FUNCTIONS

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



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

+ F2- NAMING FUNCTIONS

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



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

All function names should 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.

+ F3- NUMBER OF COLUMNS

⚠ Inside functions, the length of a line should not exceed **80 columns** (not to be confused with 80 characters!).



A tab represents 1 character, but several columns.

+ F4- NUMBER OF LINES

⚠ The body of a function should be as **short as possible**.



If the body of a function exceeds **20 lines**, it probably does too many tasks!

```
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. case-statement, where you have lots of small things for a lot of different cases, it's OK to have a longer function.

Linus Torvalds, Linux Kernel Coding Style

+ F5- ARGUMENTS

🚫 The statement of arguments should be in accordance to the **ISO/ANSI C syntax**.

A function taking no parameters should take `void` as argument in the function declaration.

```
phys_addr_t    alloc_frame();                /* F5 violation */
phys_addr_t    alloc_frame(void);            /* OK */
```

A function should not need more than **4 arguments**.

Writing variadic functions is allowed, but they should not be used to circumvent the limit of 4 parameters.

Structures should be transmitted as parameters using a **pointer, not by copy**.

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

+ F6- COMMENTS INSIDE A FUNCTION

✅ There **should be no comment** within a function.

The function should be readable and self-explanatory, without further need for explanations.



The length of a function being inversely proportional to its complexity, a complicated function should be short ; so a header comment should be enough to explain it.



L- LAYOUT INSIDE A FUNCTION SCOPE

+ L1- CODE LINE CONTENT

🚫 A line should correspond to **only one statement**.

Typical situations to avoid are:

- several assignments on the same line,
- several semi-colons on the same line, used to separate several code sequences,
- a condition and an assignment on the same line.

```
a = b = c = 0;                /* L1 violation */
a++; b++;                    /* L1 violation */
if ((ptr = malloc(sizeof(struct my_struct))) != NULL) /* L1 violation */
if (cond) return (ptr);      /* L1 violation */
a = do_something(), 5;       /* L1 violation */
```

+ L2- INDENTATION

✅ Tabulations should be 8-space-wide and used for indentation.

No space be used for indentation.

+ L3- SPACES

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

However, there will be no space between the name of a function and the opening parenthesis or after a unary operator.

All binary and ternary operators should be separated from the arguments by a space on both sides.



return is a keyword but sizeof is a unary operator!

```
return(1);                    /* L3 violation */
return (1);                  /* OK */
break;                       /* OK */
sum = term1 + 2 * term2;     /* OK */
s = sizeof(struct file);     /* OK */
```



+ L4- CURLY BRACKETS

🟢 Opening curly brackets should be **at the end of their line**, except for functions where they must be placed alone on their line.

Closing curly brackets should always be **alone on their line**, except in the case of an *else* statement.



In the case of a single-line scope, curly brackets are optional.

```
if (cond) {return (ptr);}           /* L1 & L4 violation */
while (cond) {                     /* OK */
    do_something();
}
if (cond)                         /* L4 violation */
{
    ...
} else {                          /* OK */
    ...
}
if (cond)                         /* OK */
    return (ptr);
int print_env(void)               /* OK */
{
    ...
}
int print_env(void) {             /* L4 violation */
    ...
}
```

+ L5- VARIABLE DECLARATION

🟢 Variables should be declared **at the beginning of the scope** of the function.

The for-loop counters may optionally be declared within the loop.



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

Only one variable should be declared per line.

```
long calculate_gcd(long a, long b)
{
    long biggest, smallest;        /* L5 violation */

    biggest = MAX(a, b);
    smallest = MIN(a, b);
    long rest;                    /* L5 violation */
    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 (int i = 0 ; i < max ; i++) /* OK */
        calculate_pgcd(forty_two, max);
    return (0);
}
```

+ L6- LINE JUMPS

✅ A line break should **separate the variable declarations from the remainder** of the function. No other line breaks should be present in the scope of a function.

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

    if (fd < 0)
        return (-1);
    if (fs_open(path, &fhandler)) {
        thread_free_fd(fd);
        return (-1);
    }

    thread_set_fd_handler(fd, fhandler);
    return (fd);
}
```

V- VARIABLES AND TYPES

+ V1- NAMING IDENTIFIERS

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



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

The type names defined with `typedef` should **end with `_t`**.

The names of **macros and global constants** and the content of **enums** should be written in `UPPER_CASE`.

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

+ V2- STRUCTURES

🚫 Variables could 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 {
    char *name;
    unsigned int age;
    float salary;
};

struct trashy {
    struct person player;
    unsigned int width;
    unsigned int length;
    unsigned int score;
    int i;
};
```

+ V3- POINTERS

✅ The pointer symbol (*) should be **attached to the associated variable**, with no spaces.



This rule applies only in the pointer context.

```
int* a;
int *a;
```

```
/* V3 violation */
/* OK */
```



```
int a = 3 * b;  
int strlen(char const *str);
```

```
/* OK */  
/* OK */
```



C- CONTROL STRUCTURE

Unless otherwise specified, all control structures are **allowed**.

+ C1- CONDITIONAL BRANCHING

 An `if` block **should not contain more than 3 branchings**, excluding error handling.



Instead, use an array of pointers to function or a `switch`. Take care to choose the most suitable one (you may be asked for a justification).

Nested conditonal branchings with a depth of 3 or more should be avoided.



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

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

if (....)                                /* C1 violation */
    do_something();
else if (...)
    do_something_else();
else if (...)
    do_something_more();
else
    do_one_last_thing();

while (....)                             /* OK */
    if (...)

while (....)                             /* C1 violation */
    for (...)
        if (...)
```



If ever you need to disrepect this rule, you could probably compute the **cyclomatic complexity** of your function to justify it...



+ C2- TERNARY

! Ternaries are **allowed** as far as they are kept **simple and readable**, and they do not obfuscate code.



You should never use **nested or chained ternaries**.
Ternaries should **not** be used to control program flow.

```
parity_t year_parity = (year % 2 == 0) ? EVEN : ODD;          /* OK */
return (a > 0 ? a : 0);                                       /* OK */
int a = b > 10 ? c < 20 ? 50 : 80 : e == 2 ? 4 : 8;           /* C2 violation */
already_checked ? go_there() : check();                      /* C2 violation */
```

+ C3- GOTO

✓ Your code **should not contain the goto keyword**, especially because it can very quickly participate in the creation of infamous spaghetti code, which is completely illegible.



In **rare cases**, its use makes it possible to bring readability and/or optimization to your program (error management for example).
As always, a justification will be systematically requested.



A-ADVANCED

+ A1- CONSTANT POINTERS

❗ When creating a pointer (when switching from a pointer to a parameter or internal variable of a function), if the pointed data is not modified by the function, it should be marked as **constant** (`const`).

+ A2- TYPING

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

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



`ptrdiff_t, uint8_t, int32_t,...`



H- HEADER FILES

+ H1- CONTENT

🚫 Header files should contain only:

- function prototypes,
- type declarations,
- global variable declarations,
- macros,
- static inline functions.

All these elements should be found **only** in header files.



Header files can include other headers (if and only if it is necessary).

+ H2- INCLUDE GUARD

✅ Headers should be protected from **double inclusion**.
The method and the conventions used are left free.

+ H3- MACROS

✅ Macros should **not be used for constants**.
Macros should match **only one statement**.

```
#define PI 3.14159265358979323846 /* H3 violation */
const float PI = 3.14159265358979323846; /* OK */
#define LOOP_COUNTER 0 /* H3 violation */
#define DELTA(a, b, c) ((b) * (b) - 4 * (a) * (c)) /* OK */
#define PRINT_NEXT(num) {num++; printf("%d", num);} /* H3 violation */
```