

**Epitech Documentation** 

# C Coding Style

Keep your code nice and clean



5





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:

- the organization of the delivery repository;
- the naming of identifiers;
- the overall presentation (paragraphs);
- the local presentation (lines);
- source files and headers;
- Makefiles.

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 langage 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 very 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.





Rules are categorized into 3 levels of severity: major 0, minor 0 and info 1.

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. Violating any of the major rules (even once) is a major problem and must be corrected as a priority concern.

Minor rules  $\circ$  are generally related to the visual presentation of the code, which can render the code difficult to read if not followed consistently. Repeatedly violating minor rules must be avoided, as it creates inconsistently formatted code, which in turn makes it harder to read.

**Info rules** ① are related to **specific trivial points** that are not as important as major and minor 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 and 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 infos 1.

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 the adherence of your code to a good portion of the rules. Other rules are checked manually, with the great tool that are your eyes.

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

- 🕹: the rule is completely checked by Banana;
- 🕒: the rule have 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.







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*.





# **C-O - FILES ORGANIZATION**

#### **○ J** C-O1 - CONTENTS OF THE REPOSITORY

The repository must not contain compiled (.o, .gch, .a, .so, ...), temporary or unnecessary files (\*~, #\*#, .d, toto, ...).



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**.



Beyond **5** functions in your file, you **must** subdivide your logical entity into several subentities.

## 

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 to the extent that they can significantly reduce the size of the name without losing its meaning.





# **C-G - GLOBAL SCOPE**

#### **MULTILINE STATEMENTS**

Multiline statements are allowed.

Here are examples of properly segmented multiline statements:



Do not use the backslash character ( $\setminus$ ) to break lines in C files, because it will only *visually* break the line.

As such, you will get into trouble regarding the coding style's rules!





#### ♥ J 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 C-c C-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!

##
```

#### **♥ U** C-G2 - SEPARATION OF FUNCTIONS

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

#### **○ J** 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.





#### **○ ○** 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;
int uptime = 0;
```

/\* OK \*/ /\* C-G4 violation \*/

#### **○ J C-G5** - include

include directive must only include C header (.h) files.

#### **♥ J** C-G6 - LINE ENDINGS

Line endings must be **done in UNIX style** (with  $\n$ ).



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



git config can help you keep your lines correctly ended.

#### **♥ J** C-G7 - TRAILING SPACES

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





#### **♥ O-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-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.

#### **♥ P** C-F2 - Naming Functions

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



For example, the <code>vowels\_nb()</code> and <code>dijkstra()</code> functions are incorrectly named. <code>get\_vowels\_number()</code> and <code>search\_shortest\_path()</code> 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.

## **○ J** 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.

The line break character  $(\n)$  is part of the line, and thus counts in its length.



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**.

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

#### **○ J** 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.

#### **○ J** C-F6 - FUNCTIONS WITHOUT PARAMETERS

A function taking no parameters must take void as a parameter in the function declaration.



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

#### **○** C-F7 - STRUCTURES AS PARAMETERS

Structures must be transmitted as arguments 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 violation */
{
    ...
}
```





#### **♥ J** 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 should be enough to explain it.

#### **○ J** C-F9 - NESTED FUNCTIONS

Nested functions are **not allowed**, because they are an extension of the GNU C standard, and because they greatly increase complexity.





#### C-L - LAYOUT INSIDE A FUNCTION SCOPE

#### **○ △ C-L1 - CODE LINE CONTENT**

A line must correspond to only one statement.

Typical situations to avoid include:

- several assignments on the same line;
- several semicolons on the same line used to separate several statements;
- a condition and an assignment on the same line.

The only exception to this rule is the for loop control structure, which idiomatically allows several operations on the same line (initialization, loop condition, and post-iteration operation).

#### **♥ 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), the indentation level must be incremented.

```
int main(void)
    char letter = 'H';
    int number = 14;
    if (letter == 'H') {
        my_putchar('U');
    } else if (letter == 'G') {
        if (number != 10)
           my_putchar('0');
        else {
            my_putnbr(97);
        }
    }
}
// Incorrect
int main(void)
{
int i;
```





```
// Incorrect
int main(void)
    if (true) {
    return (0);
}
// Incorrect and ugly
           int main(void)
           {
        char letter = 'H';
        int number = 14;
        if (letter == 'H') {
    my_putchar('U');
        } else if (letter == 'G') {
    if (number != 10) {
my_putchar('0');
    } else {
my_putnbr(97);
   }
        }
```

Comment blocks' bodies can be freely indented (with spaces):

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



#### ♥ J 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 parenthesis, after a unary operator, or before a semicolon.

In the precise case of a for control structure, if a semicolon inside the parentheses 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.

```
return 1;
                                   /* OK */
return (1);
                                   /* OK */
return(1);
                                   /* C-L3 violation */
return (1 + 2);
                                   /* C-L3 violation */
                                   /* OK */
break:
break ;
                                   /* C-L3 violation */
sum = term1 + 2 * term2;
                                   /* OK */
                                  /* OK */
s = sizeof(struct file);
/* C-L3 violation */
}
for (size_t i;str[i] != '\0'; i++) {    /* C-L3 violation (twice) */
}
```





#### **♥ O-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 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!

```
/* C-L1 & C-L4 violations */
if (cond) {return ptr;}
                                 /* OK */
while (cond) {
    do_something();
}
if (cond)
                                 /* C-L4 violation */
₹
                                 /* OK */
} else {
if (cond) {
                                 /* OK */
                                 /* C-L4 violation */
}
else {
}
if (cond)
                                 /* Tolerated */
   return ptr;
int print_env(void)
                                 /* OK */
{
                                /* C-L4 violation */
int print_env(void) {
    . . .
}
                                 /* OK */
struct coords {
   int x;
   int y;
};
struct coords
                                 /* C-L4 violation */
   int x;
   int y;
};
```



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 scope of the function. Only one variable must be declared per line.

The for control structures may also optionally declare variables in their initialization part.



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

```
long calculate_gcd(long a, long b)
                                       /* C-L5 violation */
    long biggest, smallest;
    biggest = MAX(a, b);
    smallest = MIN(a, b);
    long rest;
                                       /* C-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 (unsigned int i = 0; i < max; i++) {     /* OK */</pre>
        calculate_gcd(forty_two, max);
    }
    return 0;
}
```





# **♥ C-L6 - LINE JUMPS**

A line break must **separate the variable declarations from the remainder** of the function. No other line breaks must 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;
    }
        /* C-L6 violation */
    thread_set_fd_handler(fd, fhandler);
    return fd;
}</pre>
```



No line break 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 */
    1386 = 0,
    X86_64,
    ARM,
    ARM64,
    SPARC,
    POWERPC,
};
const float PI = 3.14159;
                                                             /* OK */
                                                             /* C-V1 violation */
typedef int age;
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 {
    char *name;
    unsigned int age;
    float salary;
};

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





### **♥ C-V3 - POINTERS**

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



This rule applies only in the pointer context.





# **C-C - CONTROL STRUCTURES**

Unless otherwise specified, all control structures are allowed.

#### **○ J** C-C1 - CONDITIONAL BRANCHING

A conditionnal block (while, for, if, else, ...) must not contain more than 3 branches.



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. Take care to choose the most suitable one.

**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.

```
/* OK */
if (...) {
    do_something();
} else if (...) {
    do_something_else();
} else {
    do_something_more();
if (...) {
    do_something();
} else if (...) {
    do_something_else();
} else if (...) {
    do_something_more();
                                     /* C-C1 violation */
} else {
    do_one_last_thing();
while (...) {
                                     /* OK */
    if (...) {
        do_something();
}
while (...) {
                                     /* C-C1 violation */
    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 violation */
   do_one_last_thing();
// ... is unfolded as this
if (...) {
    do_something();
} else {
   if (...) {
       do_something_else();
    } else {
       if (...) {
                                  /* C-C1 violation */
           do_one_last_thing();
        }
   }
}
// This code...
if (...) {
    do_something();
} else if (...) {
   while (...) {
                                   /* C-C1 violation */
       do_something_else();
}
// ... is unfolded as this
if (...) {
   do_something();
} else {
   if (...) {
                                   /* C-C1 violation */
       while (...) {
           do_something_else();
       }
   }
}
```



#### **○ ○ ○ C-C2 - TERNARY OPERATORS**

The use of ternary operators is **allowed as far as it is kept simple and readable**, and 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).

### **○ ○ ○ 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 illegible.





#### **C-H - HEADER FILES**

#### **№** C-H1 - CONTENT

Header files must only contain:

- function prototypes,
- type 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.

#### **○ ○** C-H2 - INCLUDE GUARD

Headers must be protected from double inclusion.

The method and the conventions used are left free.

## **○ △ C-H3 - MACROS**

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





#### **C-A - ADVANCED**

#### **1** △ 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.



Useful types include size\_t, ptrdiff\_t, uint8\_t, int32\_t, and more...

#### 

Files must end with a line break.

```
Terminal

- + x

~/Epitech Documentation> cat -e correct.c
int main(void) {$
return 0;$
}$

~/Epitech Documentation> 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 7, 2018 edition





### OC-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  ${ t static}$  keyword. It does very different things depending on where you use it.

