# Coding Style — C

version **#1.0.0**

**Assistants C/Unix 2026** <assistants@tickets.assistants.epita.fr>

## Copyright

This document is for internal use at EPITA (website) only.



**The use of this document must abide by the following rules:**

- ▷ You downloaded it from the assistants' intranet.*
- ▷ This document is strictly personal and must **not** be passed onto someone else.
- ▷ Non-compliance with these rules can lead to severe sanctions.

## Contents



*https://intra.forge.epita.fr

# 1 Introduction

## 1.1 Objectives

> "Programming style, also known as code style, is a set of rules or guidelines used when writing the source code for a computer program. It is often claimed that following a particular programming style will help programmers read and understand source code conforming to the style, and help to avoid introducing errors.
>
> The programming style used in a particular program may be derived from the coding conventions of a company or other computing organization, as well as the preferences of the author of the code."
>
> ---"Programming style" wikipedia page

## 1.2 Elements of good style

> "Good style is a subjective matter, and is difficult to define. However, there are several elements common to a large number of programming styles. The issues usually considered as part of programming style include the layout of the source code, including indentation; the use of white space around operators and keywords; the capitalization or otherwise of keywords and variable names; the style and spelling of user-defined identifiers, such as function, procedure and variable names; and the use and style of comments."
>
> ---"Programming style" wikipedia page

## 1.3 Code appearance

> "Programming styles commonly deal with the visual appearance of source code, with the goal of readability. Software has long been available that formats source code automatically, leaving coders to concentrate on naming, logic, and higher techniques. As a practical point, using a computer to format source code saves time, and it is possible to then enforce company-wide standards without debates."
>
> ---"Programming style" wikipedia page

## 1.4 Usage at EPITA

This document contains a coding style normalization, as well as recommended coding guidelines that applies in most cases. They are opinionated and they do not reflect a particular industry standard. However, we believe that a school coding style can impact future graduates and therefore companies. Most rules should be accompanied with proper rationales.

These coding rules are aimed at code consistency and readability between students, teachers and teams. This removes the freedom of how the code is formatted from the developer for the sake of greater maintenability, pedagogy and security matters. The developer can then focus his creativity on the design and on the code itself.

### 1.5 How to read this document

This standard uses the words MUST, MUST NOT, SHOULD, SHOULD NOT, MAY as described in RFC 2119.

Here are some reminders from RFC 2119:

- MUST: This word means that the definition is an absolute requirement of the specification.
- MUST NOT: This phrase means that the definition is an absolute prohibition of the specification.
- SHOULD: This word means that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighted before choosing a different course.
- SHOULD NOT: This phrase means that there may exist valid reasons in particular ciscumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighted before implementing any behavior described with this label.
- MAY: This word means that an item is truly optional. One may choose to include the item because a particular circumstance requires it or because it causes an interesting enhancement.

### 1.6 Scope of this document

This document is only about the C language.

## 2 Writing style

### 2.1 braces

When using braces for a code block, all braces MUST be on their own line.

### 2.2 braces.close

When using braces for a code block, closing braces MUST appear on the same column as the corresponding opening brace.

### 2.3 braces.single_exp

When using braces for a code block containing one single line, braces SHOULD be preferred.

```
if (var <= size)
{
    return 1;
}
```

## 2.4 braces.indent

The code between two braces MUST be indented by 4 whitespaces. Therefore, tabulations MUST NOT be used.

```
int is_odd(int var)
{
    if (var % 2 == 1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

## 2.5 braces.indent.style

The indentation style MUST be Allman (also known as “ANSI” or “BSD”).

Here are a few allowed indentation styles:

```
// Allman
if (value > max)
{
   func_one();
   func_two();
}
```

Here are a few **forbidden** indentation styles:

```
// K&R, 1TBS
if (value > max) {
   func_one();
   func_two();
}

// Whitesmith
if (value > max)
   {
   func_one();
   func_two();
   }

// Horstmann
if (value > max)
{  func_one();
   func_two();
}

// Lisp
if (value > max)
```

(continues on next page)

(continued from previous page)

```
{  func_one();
   func_two(); }
```

# 3 Global specifications

## 3.1 cast

The number of allowed explicit C cast is 0, in order to prevent you from easy mistakes, as casts might be the source of many errors. Most of the time you have an easier way to perform the cast.

Implicit casts are allowed, so this is valid:

```
void *qual;
char *acu = qual;
```

## 3.2 lines.empty

There MUST NOT be two consecutive empty lines.

## 3.3 export.other

The number of non-function exported symbol (such as a global variable) MUST be at most 1 per source file.

## 3.4 export.fun

There MUST be at most 10 exported functions per source file. In this way, source files are kept digest.

## 3.5 fun.arg.count

Functions MUST NOT take more than 4 arguments. The goal here is to increase prototype readability. Most of the time if you need to pass more arguments, you SHOULD use a structure.

### 3.6 fun.length

Functions' body SHOULD NOT contain more than 25 lines, and MUST NOT exceed 30 lines.

The enclosing braces are excluded from this count as well as the blank lines and comments.

This limit tends to make your code more digest. If you need more lines, a helper function should generally do the trick.

The following function would then have 4 counting lines:

```
int mysum(const int *arr, int len)
{
    int sum = 0;
    for (int i = 0; i < len; i++)
    {
        sum += arr[i];
    }
    return sum;
}
```

### 3.7 fun.proto.void

Prototypes MUST specify `void` if your function does not take any argument. If you do not, your function may accept a variable amount of arguments.

```
// This is valid
int function(void);

// This is not
int function();
```

### 3.8 else_if.indent

If an else contains only an `if` statement you SHOULD put the `else` and `if` keyword on the same line. Moreover, you MUST NOT indent the `if` keyword if you omit braces.

```
if (!node)
{
    return node;
}
else if (node->value) // this is correct
{
    node->value *= 2;
    return node->next;
}

if (i == 0)
{
    val = 1;
}
```

(continues on next page)

(continued from previous page)

```
else if (i == 1) // this is correct
{
    val = 8;
}
else
    if (i < 0) // this is not
        val = -1;
```

### 3.9 indent.level

Indent level SHOULD remain limited. 3 indent levels are generally sufficent and helps to respect the 80 characters per line rule.

### 3.10 fun.error

Many functions from the C library, as well as some system calls, return status values. Although special cases MUST be handled, the handling code MUST NOT clobber an algorithm. Therefore, special versions of the library or system calls, containing the error handlers, SHOULD be introduced where appropriate.

### 3.11 fun.proto.layout

Prototypes for exported function MUST appear in header files and MUST NOT appear in source files. The source file which defines an exported function MUST include the header file containing its prototype. This pervents code duplication in case you would need exported functions in several other files.

### 3.12 fun.proto.type

Prototypes MUST specify both the return type and the arguments type. This makes code easier to read and understand quickly.

```
// This is valid
int function(int param);

// This is not
int function(param);

// Neither this
function(param);
```

### 3.13 fun.ptr.call

When calling a function pointer, you MAY dereference it.

```
int fun(void (*cb)(int, int, int))
{
    (*cb)(1, 2, 3); /* good */
    cb(1, 2, 3);    /* bad */
}
```

### 3.14 fun.sep.line

Function definitions MUST be separated by one empty line.

### 3.15 gcc.builtins

You MUST NOT use gcc builtins, unless specified otherwise in the subject.

## 4 Preprocessor-level specifications

### 4.1 cpp.guard

Header files MUST be protected against multiple inclusions. You MUST use `#include` guards. In the latter case, the protection guard MUST contain the name of the file, in upper case, and with (series of) punctuation replaced with single underscores. It MAY also contain additional upper case letters and underscores. For example, if the file name is `foo++.h`, the protection key MAY contain `FOO_H`. You MUST add this protection key in a comment after the `#endif`, as follow:

```
#ifndef FOO_H
#define FOO_H

// Content of foo++.h

#endif /* ! FOO_H */
```

### 4.2 cpp.if

`#else` and `#endif` MUST be followed by a comment describing the corresponding condition. Note that the corresponding condition of the `#else` is the negation of the one of the `#if`.

### 4.3 cpp.include.filetype

Included files MUST be header files, that are, files ending with a `.h` suffix. Include of `X-Macros` is allowed from files suffixed by `.cxx`. You MUST NOT include source files.

### 4.4 cpp.include.order

Include directives SHOULD appear at the beginning of the source files, except if macro definitions change one or more headers' behavior. Those directives MUST be grouped into 3 blocks:

- The header which has the same name as the source file
- The system headers
- The local headers

Those blocks MUST be sorted in this order and MUST be separated by empty lines. Inside these blocks, headers MUST be sorted in alphabetical order.

Example for a file named *list.c*:

```
#include "list.h"

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

#include "node.h"
#include "utils.h"
```

### 4.5 cpp.mark

The preprocessor directive mark (#) MUST appear on the first column.

```
// This is valid
#include <stdio.h>

// This is not
    #include <stddef.h>
```

### 4.6 comment.lang

Comments MUST be written in the English language. They SHOULD NOT contain spelling errors, whatever language they are written in. However, omitting comments is no substitute for poor spelling abilities.

### 4.7 comment.multi

The delimiters in multi-line comments MUST appear on their own line. Intermediary lines are aligned with the delimiters and start with **.

```
/*
 * Incorrect
 */

/* Incorrect
 */
```

```
/*
** Correct
*/

/**
** Correct
*/

/* Correct */

// Correct
```

### 4.8 cpp.digraphs

Digraphs and trigraphs MUST NOT be used.

These examples are **not** valid:

```
??=include <stdio.h>

if (!node)
<%
    return NULL;
%>
```

### 4.9 cpp.macro

Macro call SHOULD NOT appear where function calls wouldn't otherwise be appropriate. Technically speaking, macro calls SHOULD parse as function calls.

### 4.10 cpp.macro.parentheses

Macro arguments SHOULD be surrounded by parentheses.

```
#define MULT(X, Y) ((X) * (Y))
```

### 4.11 cpp.macro.style

The code inside a macro definition MUST follow the specifications of the standard as a whole.

### 4.12 cpp.macro.variadic

Macros MAY be variadic.

### 4.13 cpp.token

The preprocessor MUST NOT be used to split a C keyword.

### 4.14 doc.doxygen

Documentation SHOULD be done using `doxygen`. In order to use `doxygen`, the following type of comments is allowed:

```
/**
** \brief Doxygen comment.
** \details Notice the additional '*' in the first line.
** \param first_arg Description of the param
** \return Description of the return value
*/
```

### 4.15 doc.obvious

You SHOULD NOT document the obvious.

### 4.16 doc.style

You SHOULD use the imperative when documenting, as if you were giving order to the function or entity you are describing. When describing a function, there is no need to repeat the word “function” in the documentation; the same applies obviously to any syntactic category.

# 5 Structure variables and declarations

## 5.1 decl.single

The number of declaration per line MUST be at most 1.

```
// This is not valid
int a, b;

// Here is the proper way to declare a and b
int a;
int b;
```

## 5.2 decl.vla

When declaring an array of automatic storage duration, its size specifier MUST be a constant expression, whose value can be computed at compile time. Variable-length arrays are therefore not allowed.

```
void func(int var)
{
    // Allowed
    int five[5];
    // Not allowed
    int vla[var];
}
```

## 5.3 keyword.goto

The `goto` statement MUST NOT be used.

## 5.4 stat.asm

The asm declaration MUST NOT be used.

## 5.5 enum.comma

The last line of your enumerations SHOULD end with a comma.

Example:

```
enum color
{
    RED,
    GREEN,
    BLUE,
};
```

### 5.6 ctrl.empty

To emphasize the previous rules, even single-line loops (`for` and `while`) MUST have their body on the following line. If empty, the body MUST be the `continue` statement.

Example:

```
for (; arr[i] > 0; i++)
{
    continue;
}
```

### 5.7 ctrl.switch

Incomplete switch constructs (that is, which do not cover all cases) MUST contain a default case, unless when used over an enumeration type, in which case it MUST NOT contain one. This prevents from forgetting corner cases.

The following examples are both valid:

```
void func(int value)
{
    switch (value)
    {
    case 0:
        // Do something
        break;
    case 1:
        // Do something else
        break;
    default:
        // For all other cases
        break;
    }
}
```

```
enum color
{
    red,
    green,
    blue,
};

void func(enum color c)
{
    switch (c)
    {
    case red:
        // Red case related code
        break;
    case green:
        // Green case related code
        break;
```

(continues on next page)

(continued from previous page)

```
    case blue:
        // Blue case related code
        break;
    }
}
```

### 5.8 ctrl.switch.fallthrough

If you have good reasons to do so, you MAY omit the break at the end of a switch case. In that case, you MUST add a *FALLTHROUGH* comment.

Example:

```
switch (c)
{
case 'a':
    i++;
/* FALLTHROUGH */
case 'b':
    puts("This is a letter");
    break;
}
```

### 5.9 ctrl.switch.indentation

Each case conditional MUST be on the same column as the construct's opening brace. The code associated with the case conditional MUST be indented from the case.

The following example is valid:

```
switch (len)
{
case 0:
    // Associated code
    break;
case 1:
    // Associated code
    break;
...
default:
    // Every other case associated code
    break;
}
```

### 5.10 ctrl.switch.padding

There MUST NOT be any whitespace between a label and the following colon (`:`), or between the `default` keyword and the following colon.

```
switch (value)
{
// This is valid
case 0:
    break;
// This is not valid
case 1 :
    break;
}
```

### 5.11 decl.point

The pointer symbol (`*`) in declarations MUST appear next to the variable name, not next to the type.

```
// This is valid
int *ptr_a = &a;

// This is not
int* ptr_b = &b;
```

### 5.12 exp.align

Parenthesized expressions and argument lists that span over multiple lines MUST be indented so that each continuation line is aligned with the opening parenthesis.

```
// This is valid
if (longConditionPartOne && longConditionPartTwo && longConditionPartThree
    && longConditionPartFour)
{
    return 1;
}

// These are not
if (longConditionPartOne && longConditionPartTwo && longConditionPartThree &&␣
↪longConditionPartFour)
{
    return 1;
}
if (longConditionPartOne && longConditionPartTwo && longConditionPartThree
  && longConditionPartFour)
{
    return 1;
}
```

### 5.13 exp.linebreak

Expressions MAY span over multiple lines. When a line break occurs within an expression, it MUST appear just before a binary operator, in which case the binary operator MUST be indented with at least an extra indentation level.

```
int very_long_var = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 // etc
    + 13 + 14 + 15;
```

### 5.14 exp.padding

All binary and ternary operators MUST be padded on the left and right by one space, including assignment operators. Prefix and suffix operators MUST NOT be padded, neither on the left nor on the right. When necessary, padding is done with a single whitespace.

### 5.15 comma

The comma MUST be followed by a single space, except when they separate arguments in function (or macro) calls and declarations and the argument list spans multiple lines: in such cases, there MUST NOT be any trailing whitespace at the end of each line.

```
// These are valid
int func(int a, int b);
int array[] = {
    0,
    1,
};

// These are not
int func(int a,int b);
int func(int a ,int b);
```

### 5.16 ctrl.do_while

The closing brace of the `do while` control structure MUST be on the same line as the `while` keyword.

This rule is an exception of the `braces` rule.

```
int var = 0;
do
{
    ++var;
} while (var < 10);
```

### 5.17 ctrl.for

Multiple statements MAY appear in the initial and iteration parts of the `for` structure. For the effect, commas may be used to separate statements.

This rule is an exception to the `decl.single` rule.

```
// This is valid
for (int i = 0, j = 0; i < 5 || j < 6; i++, j++)
{
    printf("i = %d, j = %d\n", i, j);
}
```

### 5.18 ctrl.indentation

The code following a control structure MUST be indented. The number of whitespaces MUST be 4. If the code following a control structure is composed of only one expression, you MAY omit the braces, as follow:

```
if (var < 0)
    return -var;
else
    return var;
```

### 5.19 ctrl.switch.control

Control structure MUST NOT span over several case blocks.

This is **not** valid:

```
switch (i)
{
case 0:
    puts("toto");
    for (int i = 0; i < 10; i++)
    {
    case 1:
        puts("tata");
    }
    break;
default:
    puts("titi");
    break;
}
```

## 5.20 decl.init

Variables SHOULD be initialized at the point of declaration by expressions that are valid according to the rest of the coding style.

This is bad:

```
char c = (str++, *str);
int i = tab<:c:>;
```

These declarations are correct:

```
int bar = 3 + 6;
char *opt = argv[cur];
unsigned int *foo = &bar;
unsigned int baz = 1;
int *p = NULL;
char *opt = argv[2];
int mho = CALL(x);
size_t foo = strlen("bar");
```

## 5.21 decl.init.multiline

When initializing a local structure (a C99 feature) or a static array, the initializer value MAY start on the same line as the declaration.

If the initialization is too long and exceed the column limitation, the initializer value MUST start on the line after the declaration. In this case, each brace must be on their own line.

This rule is an exception of the `braces` rule.

```
// Correct
int array[5] = { 0 };

// Correct
int array[5] = { 1, 2, 3 };

// Incorrect
int array[5] =
{ 1, 2, 3 };

// Correct
int array[2][2] = {
    { 0, 0 }, //
    { 0, 0 } //
};
```

### 5.22 exp.args

There MUST NOT be any whitespace between the function name and the opening parenthesis for arguments, either in declarations or calls.

```
// This declaration is valid
int func(void);

// This one is not
int func (void);

int main(void)
{
    puts("This call is valid");
    puts ("This one is not");
}
```

### 5.23 keyword.arg

*Functional* keyword MUST have their argument(s) enclosed between parentheses. Especially note that `sizeof` is a keyword, while `exit` is not.

### 5.24 semicolon

The semicolon MUST NOT be preceded by a whitespace, except if alone on its line. However, a semicolon SHOULD NOT be alone on its line. If the semicolon is followed by a non-empty statement, a single whitespace MUST follow the semicolon.

```
puts("This semicolon is valid");

puts("This one is not") ;

// This for statement is valid
for (char c = 'a'; c < 'f'; c++)
{
    putc(c);
}

// This one is not
for (char c = 'a';c < 'f';c++)
{
    putc(c);
}
```

### 5.25 stat.sep

Comma operator MUST NOT be used outside the `for` structure.

This is allowed:

```
for (int i = 0; i < 8; i++, c++)
{...}
```

But this is forbidden:

```
int a = puts("message"), 12;
```

## 6 File

### 6.1 file.deadcode

In order to disable large amounts of code, you SHOULD NOT use comments. Use `#if 0` and `#endif` instead. Delivered project sources SHOULD NOT contain disabled code blocks.

Of course, rationale C comments do remain.

```
// This is an allowed comment to describe something.

/**
** This is a deadcode. Commented codes are deadcodes.
**
** int main(void)
** {
**     return 0;
** }
*/
```

### 6.2 file.dos

The DOS CR+LF line terminator MUST NOT be used.

### 6.3 file.spurious

There MUST NOT be any blank lines at the beginning or the end of the file.

### 6.4 file.terminate

The last character of the file MUST be a line feed.

### 6.5 file.trailing

There MUST NOT be any trailing whitespace at the end of the line.

## 7 Naming conventions

### 7.1 name.abbr

Names MAY be abbreviated but only when it allows for shorter code without loss of meaning. Names SHOULD even be abbreviated when long standing programming practices allow so.

### 7.2 name.case

Variable names, C function names and file names MAY be expressed using lower case letters, digits and underscores only. More precisely entity names SHOULD be matched by the following regular expression:

```
[a-z][a-z0-9_]*
```

### 7.3 name.case.macro

Macro names MUST be upper case. Macro arguments MUST be in title case.

```
#define MAX(Left, Right) ((Left) > (Right) ? (Left) : (Right))
```

### 7.4 name.gen

Entities (variables, functions, macros, types, files or directories) SHOULD have explicit and/or mnemonic names.

### 7.5 name.fun

Function names SHOULD always use a verb.

### 7.6 name.lang

Names MUST be expressed in English. Names SHOULD be expressed in correct English, i.e. without spelling mistakes.

### 7.7 name.prefix.global

Global variable identifiers (variable names in the global scope) when allowed/used MUST start with `g_`.

```
extern int g_debug;
```

### 7.8 name.sep

Composite names MUST be separated by underscores ('_').

### 7.9 name.struct

Structure and union names MUST NOT be aliased using `typedef`.

**Be careful!**

`typedef` may only be used for function pointers.

## 8 Project

### 8.1 mk

The input file for the `make` command MUST be named `Makefile`, with a **capital M**.

### 8.2 mk.rules.clean

The `clean` rule SHOULD remove the targets of compilation (such as executable binaries, shared objects, library archives, `.pdf` and `.html` manuals, etc.). In other words, it removes everything that has been generated from your `Makefile`.

## 8.3 proj.directory

Each project sources MUST be delivered in a directory, the name of which shall be announced in advance by the assistants.

## 8.4 proj.mk.flags

C sources MUST compile without warnings when using strict compilers. The GNU C compiler, when provided with strict warning options is considered a strict compiler for this purpose. Especially when GCC is available as a standard compiler on a system, source code MUST compile with GCC and the following options:

```
-std=c99 -pedantic -Werror -Wall -Wextra -Wvla
```

*We've always defined ourselves by the ability to overcome the impossible.*