Coding Style — C

version **#v2.2.3**

Assistant C/UNIX 2021 <assistants@tickets.assistants.epita.fr>

Copyright

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

Contents

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

1 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.

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.indent

The between two braces MUST be indented by 4 spaces. As a reminder of a previous rule, tabulations MUST NOT be used.

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

2.4 braces.indent.style

The indentation style MUST be Allman (also known as “ANSI” or “BSD”) , and MUST NOT be any one of the following: K&R, 1TBS, Whitesmith, Horstmann, Lisp, GNU.

Here is the allowed indentation style:

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

Here is the **not** allowed 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)
{  func_one();
   func_two(); }
```

3 Global specifications

3.1 cast

The number of allowed explicit C cast is 0.

Implicit casts are allowed, so this is valid:

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

3.2 export.other

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

3.3 export.fun

There MUST be at most 5 exported functions per source file.

3.4 file.fun.count

There MUST NOT appear more than 10 functions (exported + local) per source file.

3.5 fun.arg.count

Functions MUST NOT take more than 4 arguments.

3.6 fun.length

Functions' body MUST NOT contain more than 25 lines (50 lines in c++).

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

The following function would then have 4 counting lines:

```
int mysum(const int *arr, size_t len)
{
   int sum = 0;
   for (size_t 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.

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 (!node)
    return NULL;
else if (node->next) // this is correct
    return node->next;
else
    if (node->value) // this is not
        return node;
```

3.9 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.

3.10 fun.proto

Any exported function MUST be properly prototyped.

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.

3.12 fun.proto.type

Prototypes MUST specify both the return type and the arguments type.

4 Preprocessor-level specifications

4.1 cpp.guard

Header files MUST be protected against multiple inclusions. You MAY use the `#pragma once` directive in order to achieve this. Otherwise, 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 if `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 that of the `#if`.

4.3 cpp.include.filetype

Included files MUST be header files, that are, files ending with a `.h` (a `.hh` or a `.hxx` for C++) suffix. You MUST NOT include source files. In order to use X-Macros, you MAY include `.def` files in your code.

4.4 cpp.mark

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

4.5 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.

4.6 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
```

4.7 cpp.digraphs

Digraphs and trigraphs MUST NOT be used.

4.8 cpp.macro

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

4.9 cpp.macro.parentheses

Macro arguments SHOULD be surrounded by parentheses.

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

4.10 cpp.macro.style

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

4.11 cpp.macro.variadic

Macros MAY be variadic.

4.12 cpp.token

The preprocessor MUST NOT be used to split a token.

4.13 doc.doxygen

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

```
/**
** Doxygen comment.
** Notice the additional '*' in the first line.
*/
```

4.14 doc.obvious

You SHOULD NOT document the obvious.

4.15 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 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 SHOULD NOT contain one.

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

5.2 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:
```

(continues on next page)

(continued from previous page)

```
    // Associated code
    break;
...
default:
    // Every other case associated code
    break;
}
```

5.3 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.

5.4 decl.point

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

```
void func(const char *str)
{

}
```

5.5 decl.single

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

5.6 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];
}
```

5.7 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.

5.8 keyword.goto

The `goto` statement MUST NOT be used.

5.9 stat.asm

The asm declaration MUST NOT be used.

5.10 ctrl.empty

To emphasize the previous rules, even single-line loops (`for` and `while`) MUST have their body on the following line. This MAY be a single semicolon, but SHOULD be the `continue` statement.

5.11 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 + 13 // etc
    + 14 + 15;
```

5.12 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.

5.13 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);
```

5.14 ctrl.for

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

5.15 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;
```

5.16 ctrl.switch.control

Control structure MUST NOT span over several case blocks.

5.17 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");
```

5.18 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 } //
};
```

5.19 decl.type

When negative values are impossible, you MUST specify unsigned. Appropriate `typedef` (such as `size_t`) SHOULD be used when available.

5.20 exp.args

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

5.21 keyword.arg

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

5.22 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.

5.23 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 not rest.

```
// 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 line of this file MUST NOT be a single line feed.

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.lang

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

7.6 name.prefix.global

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

```
extern int g_debug;
```

7.7 name.sep

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

7.8 name.struct

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

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:

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

The only way out is through