ETNA Documentation

C Coding Style

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

It concerns:

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

Warning

- 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 ETNA projects, regardless of the year or unit. It applies to all source files (`.c`) and headers (`.h`), as well as Makefiles.

Warning

- 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.

Informations

- 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.
- ...

Informations

- 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, code expressiveness, flexibility, optimization, etc.), you may infringe some of the *Coding Style* points.

Warning

- 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, please 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.

Warning

- Implicit rules are considered as infos .

Informations

- 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**.

Informations

- 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.

Warning

- Beyond 5 non-static 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**.

Informations

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

Informations

- Abbreviations are tolerated if they are guaranteed to be understood by any possible reader. In doubt, be as explicit as possible.

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.

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

G2 - Separation of functions

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

G3 - Global variables

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

Warning

- Non-constant static variables in the global scope are tolerated but should not be abused.
- Non-constant non-static variables in the global scope should be **avoided** as much as possible.
- A variable is called constant if and only if it is correctly marked as such with the keyword **const**. Watch out, this keyword follows some particular and sometimes surprising rules!

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

G4 - Static

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

Warning

- Be careful not to confuse the different uses of the `static` keyword.

F - Functions

F1 - Coherence of functions

A function should only perform one simple task. This allows greater flexibility.

Informations

- 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**.

Informations

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

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

Informations

- Abbreviations are tolerated if they are guaranteed to be understood by any possible reader. In doubt, be as explicit as possible.

Informations

If you write libraries or utility code, it is better to adopt a naming convention such as `mylib_myfunction()`

F3 - Number of columns

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

Warning

- A tabulation represents **1 character**, but several columns.

F4 - Number of lines

The body of a function should contain the code needed to perform its purpose.

Warning

- 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.
Within 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 follow 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 coffee_machine *machine, int nb_cups)  /* OK */
{
    ...
}
```

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.

Informations

- 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

4 spaces should be used for indentation. Tabulations should never be used for indenting purposes.

L3 - Spaces

Always place a **space after a comma or a keyword** (with or without argument).

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.

Warning

- `return` is a keyword but `sizeof` is a unary operator!

```
return(1);                  /* L3 violation */
return (1);                 /* 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.

Informations

- 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 - Variables declaration

Variables should be declared **at the beginning of a scope** (it can be either a function scope, an if-block subscope, etc.). Variables scopes should be as tight as possible.

Informations

- Nothing prevents you from declaring and assigning a variable on the same line. It is even encouraged, as it ensures initialization of the variable.

Only one variable should be declared per line.

```
long calculate_pgcd(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 */
        long result = calculate_pgcd(forty_two, max); /* OK */
    }
    return (0);
}
```

L6 - Line jumps

A line break should **separate the variable declarations from the remainder** of the function. Single line breaks are tolerated inside a function if they help with readability

```
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);
    }
                                        /* Tolerated */
    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 only of lowercase,

numbers, and underscores).

Informations

- 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 the content of enums should be written in** `UPPER_CASE`.

```
#define IS_PAGE_ALIGNED(x) (!((x) & (PAGE_SIZE - 1)))       /* OK */
enum arch {                                                 /* OK */
    I386 = 0,
    X86_64,
    ARM,
    ARM64,
    SPARC,
    POWERPC,
};

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

struct trashy {             /* V2 violation */
    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.

Warning

- **This rule applies only in the pointer context.**

```
int* a;                         /* V3 violation */
int *a;                         /* OK */
int a = 3 * b;                  /* OK */
int strlen(char const *str);    /* OK*/
```

C - Control structure

Unless otherwise specified, all control structures are allowed.

C1 - Ternary

Ternaries are allowed as far as they are kept simple and readable, and they do not obfuscate code. Remember that ternaries are expressions: use their result.

Warning

- **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;         /* C1 violation */
already_checked ? go_there() : check();                     /* C1 violation */
```

C2 - 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.

Informations

- **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, if the pointed data is not modified throught the pointer, it should be marked as constant (`const`). This rule applies for both pointer parameters and regular pointer variables.

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 */
```

Informations

- `ptrdiff_t`, `uint8_t`, `int32_t`, ...

H - Header files

H1 - Content

Header files should contain only:

- Prototypes of functions.
- Type declarations.
- Global variable declarations.
- Macros.
- Static inline functions.

All these elements should be found only in header files.

Informations

- 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 match only one statement.

```
#define DELTA(a, b, c)      ((b) * (b) - 4 * (a) * (c))     /* OK */
#define PRINT_NEXT(num)     {num++; printf("%d", num);}     /* H3 violation */
```