The Norm

Version 4.1

Summary: This document describes the applicable standard (Norm) at 42: a programming standard that defines a set of rules to follow when writing code. The Norm applies to all C projects within the Common Core by default, and to any project where it's specified.

Contents

Chapter I

Foreword

The `norminette` is a Python and open source code that checks Norm compliance of your source code. It checks many constraints of the Norm, but not all of them (eg. subjective constraints). Unless specific local regulations on your campus, the `norminette` prevails during evaluations on the controlled items. In the following pages, rules that are not checked by the `norminette` are marked with *(*)*, and can lead to project failure (using the Norm flag) if discovered by the evaluator during a code review.

Its repository is available at https://github.com/42School/norminette.

Pull requests, suggestions and issues are welcome!

Chapter II

Why?

The Norm has been carefully crafted to fulfill many pedagogical needs. Here are the most important reasons for all the choices below:

- Sequencing: coding implies splitting a big and complex task into a long series of elementary instructions. All these instructions will be executed in sequence: one after another. A beginner that starts creating software needs a simple and clear architecture for their project, with a full understanding of all individual instructions and the precise order of execution. Cryptic language syntaxes that do multiple instructions apparently at the same time are confusing, functions that try to address multiple tasks mixed in the same portion of code are source of errors.
  The Norm asks you to create simple pieces of code, where the unique task of each piece can be clearly understood and verified, and where the sequence of all the executed instructions leaves no doubt. That's why we ask for 25 lines maximum in functions, also why `for`, `do .. while`, or ternaries are forbidden.

- Look and Feel: while exchanging with your friends and workmates during the normal peer-learning process, and also during the peer-evaluations, you do not want to spend time to decrypt their code, but directly talk about the logic of the piece of code.
  The Norm asks you to use a specific look and feel, providing instructions for the naming of the functions and variables, indentation, brace rules, tab and spaces at many places... . This will allow you to smoothly have a look at other's codes that will look familiar, and get directly to the point instead of spending time reading the code before understanding it. The Norm also comes as a trademark. As part of the 42 community, you will be able to recognize code written by another 42 student or alumni when you'll be in the labor market.

- Long-term vision: making the effort to write understandable code is the best way to maintain it. Each time that someone else, including you, has to fix a bug or add a new feature they won't have to lose their precious time trying to figure out what it does if previously you did things in the right way. This will avoid situations where pieces of code stop being maintained just because it is time-consuming, and that can make the difference when we talk about having a successful product in the market. The sooner you learn to do so, the better.

- References: you may think that some, or all, the rules included on the Norm are arbitrary, but we actually thought and read about what to do and how to do it.

We highly encourage you to Google why the functions should be short and just do one thing, why the name of the variables should make sense, why lines shouldn't be longer than 80 columns wide, why a function should not take many parameters, why comments should be useful, etc.

Chapter III

The Norm

III.1 Naming

- A structure's name must start by `s_`.
- A typedef's name must start by `t_`.
- A union's name must start by `u_`.
- An enum's name must start by `e_`.
- A global's name must start by `g_`.
- Identifiers, like variables, functions names, user defined types, can only contain lowercases, digits and '_' (snake_case). No capital letters are allowed.
- Files and directories names can only contain lowercases, digits and '_' (snake_case).
- Characters that aren't part of the standard ASCII table are forbidden, except inside litteral strings and chars.
- *(*)* All identifiers (functions, types, variables, etc.) names should be explicit, or a mnemonic, should be readable in English, with each word separated by an underscore. This applies to macros, filenames and directories as well.
- Using global variables that are not marked const or static is forbidden and is considered a norm error, unless the project explicitly allows them.
- The file must compile. A file that doesn't compile isn't expected to pass the Norm.

III.2 Formatting

- Each function must be at most 25 lines long, not counting the function's own braces.
- Each line must be at most 80 columns wide, comments included. Warning: a tabulation doesn't count as a single column, but as the number of spaces it represents.
- Functions must be separated by an empty line. Comments or preprocessor instructions can be inserted between functions. At least an empty line must exists.
- You must indent your code with 4-char-long tabulations. This is not the same as 4 spaces, we're talking about real tabulations here (ASCII char number 9). Check that your code editor is correctly configured in order to visually get a proper indentation that will be validated by the `norminette`.
- Blocks within braces must be indented. Braces are alone on their own line, except in declaration of struct, enum, union.
- An empty line must be empty: no spaces or tabulations.
- A line can never end with spaces or tabulations.
- You can never have two consecutive empty lines. You can never have two consecutive spaces.
- Declarations must be at the beginning of a function.
- All variable names must be indented on the same column in their scope. Note: types are already indented by the containing block.
- The asterisks that go with pointers must be stuck to variable names.
- One single variable declaration per line.
- Declaration and an initialisation cannot be on the same line, except for global variables (when allowed), static variables, and constants.
- In a function, you must place an empty line between variable declarations and the remaining of the function. No other empty lines are allowed in a function.
- Only one instruction or control structure per line is allowed. Eg.: Assignment in a control structure is forbidden, two or multiple assignments on the same line is forbidden, a newline is needed at the end of a control structure, ...
- An instruction or control structure can be split into multiple lines when needed. The following lines created must be indented compared to the first line, natural spaces will be used to cut the line, and if applies, operators will be at the beginning of the new line and not at the end of the previous one.
- Unless it's the end of a line, each comma or semi-colon must be followed by a space.
- Each operator or operand must be separated by one - and only one - space.
- Each C keyword must be followed by a space, except for keywords for types (such as int, char, float, etc.), as well as sizeof.

- Control structures (if, while..) must use braces, unless they contain a single instruction on a single line.

General example:

```
int             g_global;
typedef struct  s_struct
{
    char    *my_string;
    int     i;
}               t_struct;
struct          s_other_struct;

int     main(void)
{
    int     i;
    char    c;

    return (i);
}
```

III.3 Functions

- A function can take 4 named parameters at most.
- A function that doesn't take arguments must be explicitly prototyped with the word "void" as the argument.
- Parameters in functions' prototypes must be named.
- You can't declare more than 5 variables per function.
- Return of a function has to be between parenthesis, unless the function returns nothing.
- Each function must have a single tabulation between its return type and its name.

```
int my_func(int arg1, char arg2, char *arg3)
{
    return (my_val);
}

int func2(void)
{
    return ;
}
```

III.4 Typedef, struct, enum and union

- As other C keywords, add a space between “struct” and the name when declaring a struct. Same applies to enum and union.
- When declaring a variable of type struct, apply the usual indentation for the name of the variable. Same applies to enum and union.
- Inside the braces of the struct, enum, union, regular indentation rules apply, like any other blocks.
- As other C keywords, add a space after “typedef”, and apply regular indentation for the new defined name.
- You must indent all structures’ names on the same column for their scope.
- You cannot declare a structure in a .c file.

III.5 Headers - a.k.a include files

- *(*)* The allowed elements of a header file are: header inclusions (system or not), declarations, defines, prototypes and macros.
- All includes must be at the beginning of the file.
- You cannot include a C file in a header file or another C file.
- Header files must be protected from double inclusions. If the file is `ft_foo.h`, its bystander macro is `FT_FOO_H`.
- *(*)* Inclusion of unused headers is forbidden.
- Header inclusion can be justified in the .c file and in the .h file itself using comments.

```
#ifndef FT_HEADER_H
# define FT_HEADER_H
# include <stdlib.h>
# include <stdio.h>
# define FOO "bar"

int     g_variable;
struct  s_struct;

#endif
```

III.6 The 42 header - a.k.a start a file with style

- Every .c and .h file must immediately begin with the standard 42 header: a multi-line comment with a special format including useful informations. The standard header is naturally available on computers in clusters for various text editors (emacs: using `C-c C-h`, vim using `:Stdheader` or `F1`, etc...).
- *(*)* The 42 header must contain several informations up-to-date, including the creator with login and student email (@student.campus), the date of creation, the login and date of the last update. Each time the file is saved on disk, the information should be automatically updated.

`The default standard header may not automatically be configured with your personnal information. You may need to change it to follow the previous rule.`

III.7 Macros and Pre-processors

- *(*)* Preprocessor constants (or #define) you create must be used only for literal and constant values.
- *(*)* All #define created to bypass the norm and/or obfuscate code are forbidden.
- *(*)* You can use macros available in standard libraries, only if those ones are allowed in the scope of the given project.
- Multiline macros are forbidden.
- Macro names must be all uppercase.
- You must indent preprocessor directives inside #if, #ifdef or #ifndef blocks.
- Preprocessor instructions are forbidden outside of global scope.

III.8 Forbidden stuff!

- You're not allowed to use:
  - for
  - do...while
  - switch
  - case
  - goto
- Ternary operators such as '?'.
- VLAs - Variable Length Arrays.
- Implicit type in variable declarations

```
int main(int argc, char **argv)
{
    int     i;
    char    str[argc]; // This is a VLA

    i = argc > 5 ? 0 : 1 // Ternary
}
```

III.9 Comments

- Comments cannot be inside function bodies. Comments must be at the end of a line, or on their own line
- *(*)* Your comments should be in English, and useful.
- *(*)* A comment cannot justify the creation of a carryall or bad function.

A carryall or bad function usually comes with names that are not explicit such as f1, f2... for the function and a, b, c,.. for the variables names. A function whose only goal is to avoid the norm, without a unique logical purpose, is also considered as a bad function. Please remind that it is desirable to have clear and readable functions that achieve a clear and simple task each. Avoid any code obfuscation techniques, such as the one-liner, ...

III.10 Files

- You cannot include a .c file in a .c file.
- You cannot have more than 5 function-definitions in a .c file.

III.11 Makefile

Makefiles aren't checked by the `norminette`, and must be checked during evaluation by the student when asked by the evaluation guidelines. Unless specific instructions, the following rules apply to the Makefiles:

- The *$(NAME)*, *clean*, *fclean*, *re* and *all* rules are mandatory. The *all* rule must be the default one and executed when typing just `make`.
- If the makefile relinks when not necessary, the project will be considered non-functional.
- In the case of a multibinary project, in addition to the above rules, you must have a rule for each binary (eg: $(NAME_1), $(NAME_2), ...). The "all" rule will compile all the binaries, using each binary rule.
- In the case of a project that calls a function from a non-system library (e.g.: `libft`) that exists along your source code, your makefile must compile this library automatically.
- All source files needed to compile your project must be explicitly named in your Makefile. Eg: no "*.c", no "*.o" , etc ...