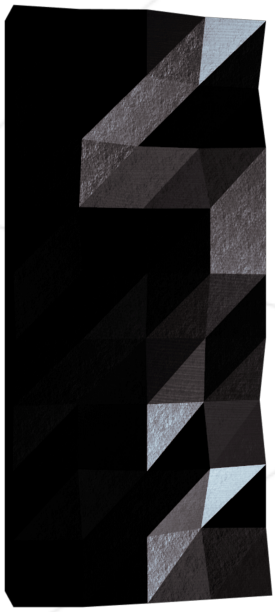

The Norm

Version 2.0.2

Summary: This document describes the applicable standard (Norm) at 42. A programming standard defines a set of rules to follow when writing code. You must always respect the Norm for all C projects at the school, unless otherwise specified.

Contents

Chapter I

Foreword

This document describes the applicable standard (Norm) at 42. A programming standard defines a set of rules to follow when writing code. You must always respect the Norm for all C projects at the school, unless otherwise specified.

I.1 Why impose a standard?

The Norm's two main objective : 1. To format and standardize your code so that anyone (students, staff and even yourself) can read and understand them easily. 2. To guide you in writing short and simple code.

I.2 The Norm for submissions

All of your C files must respect the school's Norm. It will be checked by your grader. If you made any Norm error you'll get a 0 for the exercise or even for the whole project. During peer-evaluations, your grader will have to launch the "Norminette" present in your submission's dumps. Only the mandatory part of the Norm will be checked by the "Norminette".

I.3 Suggestions

You'll realise soon enough that the Norm isn't as intimidating as it seems. On the contrary, it'll help you more than you know. It'll allow you to read your classmates' code more easily and vice versa. A source file containing one Norm error will be treated the same way as a source file containing 10 Norm errors. We strongly advise you to keep the Norm in mind while coding - even though you may feel it's slowing you down at first. In time, it'll become a reflex.

I.4 Disclaimers

"Norminette" is a program, and all programs are subject to bugs. Should you spot one, please report it in the forum's appropriate section. However, as the "Norminette" always prevails, all your submissions must adapt to its bugs.

Chapter II

The Norm

II.1 Denomination

Mandatory part

- 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_`.
- Variables and functions names can only contain lowercases, digits and '_' (Unix Case).
- Files and directories names can only contain lowercases, digits and '_' (Unix Case).
- The file must compile.
- Characters that aren't part of the standard ascii table are forbidden.

Advice part

- Objects (variables, functions, macros, types, files or directories) must have the most explicit or most mnemonic names as possible. Only 'counters' can be named to your liking.
- Abreviations are tolerated as long as it's to shorten the original name, and that it remains intelligible. If the name contains more than one word, words shall be separated by '_'.
- All identifiers (functions, macros, types, variables, etc) must be in English.
- Any use of global variable must be justifiable.

II.2 Formatting

Mandatory part

- All your files must begin with the standard school header (from the first line of the file). This header is available by default with `emacs` and `vim` in the dumps.
- You must indent your code with 4-space tabulations. This is not the same as 4 average spaces, we're talking about real tabulations here.
- Each function must be maximum 25 lines, not counting the function's own curly brackets.
- Each line must be at most 80 columns wide, comments included. Warning : a tabulation doesn't count as a column, but as the number of spaces it represents.
- One instruction per line.
- An empty line must be empty: no spaces or tabulations.
- A line can never end with spaces or tabulations.
- You need to start a new line after each curly bracket or end of control structure.
- Unless it's the end of a line, each comma or semi-colon must be followed by a space.
- Each operator (binary or ternary) 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.
- Each variable declaration must be indented on the same column.
- The asterisks that go with pointers must be stuck to variable names.
- One single variable declaration per line.
- We cannot stick a declaration and an initialisation on the same line, except for global variables and static variables.
- Declarations must be at the beginning of a function, and must be separated by an empty line.
- There cannot be an empty line between declarations or implementations.
- Multiple assignments are strictly forbidden.
- You may add a new line after an instruction or control structure, but you'll have to add an indentation with brackets or affectation operator. Operators must be at the beginning of a line.

II.3 Functions parameters

Mandatory part

- A function can take 4 named parameters maximum.
- A function that doesn't take arguments must be explicitely pototyped with the word "void" as argument.

II.4 Functions

Mandatory part

- Parameters in functions' prototypes must be named.
- Each function must be separated from the next by an empty line.
- You can't declare more than 5 variables per bloc.
- Return of a function has to be between parantheses.

Advice part

- Your functions' identifiers must be aligned within a same file. Same goes for header files.

II.5 Typedef, struct, enum and union

Mandatory Part

- Add a tabulation when declaring a struct, enum or union.
- When declaring a variable of type struct, enum or union, add a single space in the type.
- Add a tabulation between two parameters of a typedef.
- When declaring a struct, union or enum with a typedef, all rules apply. You must align the typedef's name with the struct/union/enum's name.
- You cannot declare a structure in a .c file.

II.6 Headers

Mandatory Part

- The things allowed in header files are : header inclusions (system or not), declarations, defines, prototypes and macros.
- All includes (.c or .h) must be at the beginning of the file.

- We'll protect headers from double inclusions. If the file is `ft_foo.h`, its bystander macro is `FT_FOO_H`.
- Functions' prototypes must exculively be in .h files.
- Unused header inclusions (.h) are forbidden.

Advice part

- All header inclusions must be justified in a .c file as well as in a .h file.

II.7 Macros and Pre-processors

Mandatory part

- Preprocessor constants (or #define) you create must be used only for associate literal and constant values.
- All #define created to bypass the norm and/or obfuscate code are forbidden. This point must be checked by a human.
- 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.
- Only macros names are uppercase.
- You must indent characters following #if , #ifdef or #ifndef.

II.8 Forbidden stuff !

Mandatory part

- You're not allowed to use :
  - for
  - do...while
  - switch
  - case
  - goto
- Nested ternary operators such as '?'.
- VLAs - Variable Length Arrays.

II.9 Comments

Mandatory part

- You're allowed to comment your code in your source files.
- Comments cannot be inside functions' bodies.
- Comments start and end by a single line. All intermediary lines must align and start by '**'.
- No comments with //.

Advice part

- You comments must be in English. And they must be useful.
- A comment cannot justify a "bastard" function.

II.10 Files

Mandatory part

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

II.11 Makefile

Mandatory part

- The $(NAME), clean, fclean, re and all rules are mandatory.
- If the makefile relinks, the project will be considered non-functional.
- In the case of a multibinary project, on top of the rules we've seen, you must have a rule that compiles both binaries as well as a specific rule for each binary compiled.
- In the case of a project that calls a functions library (e.g.: `libft`), your makefile must compile this library automatically.
- All source files you need to compile your project must be explicitly named in your Makefile.