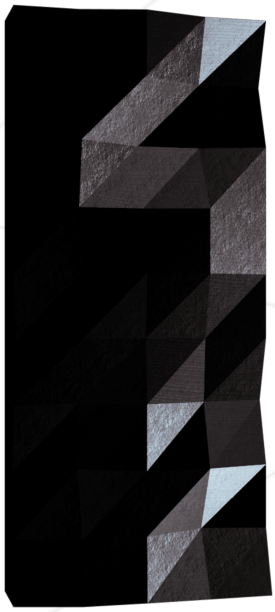

La Norme

Version 4

Résumé: Ce document décrit La Norme C en vigueur à 42. Une norme de programmation définit un ensemble de règles régissant l'écriture d'un code. La Norme s'applique par défaut à tous les projets C du Cercle Intérieur, et à tout projet où elle est spécifiée.

Table des matières

Chapitre I

Avant-propos

La Norminette est en Python et est open source.
Vous pouvez en consulter les sources ici : https ://github.com/42School/norminette.
Les Pull Requests, suggestions et Issues sont les bienvenues !

Chapitre II

Pourquoi ?

La norme a été soigneusement conçue pour répondre à de nombreux besoins pédagogiques. Voici les raisons les plus importantes qui justifient tous les choix ci-dessous :

- Sequencing : le codage implique la division d'une tâche importante et complexe en une longue série d'instructions élémentaires. Toutes ces instructions seront exécutées en séquence : l'une après l'autre. Un débutant qui commence à créer un logiciel a besoin d'une architecture simple et claire pour son projet, avec une compréhension totale de toutes les instructions individuelles et l'ordre précis d'exécution. Les syntaxes de langage cryptiques qui exécutent plusieurs instructions apparemment en même temps sont déroutantes, les fonctions qui tentent d'aborder plusieurs tâches mélangées dans la même portion de code sont sources d'erreurs. La norme vous demande de créer des morceaux de code simples, où la tâche unique de chaque morceau peut être clairement comprise et vérifiée, et où la séquence de toutes les instructions exécutées ne laisse aucun doute. C'est pourquoi nous demandons un maximum de 25 lignes pour les fonctions, et aussi pourquoi `for`, `do .. while`, ou les ternaires sont interdits.

- L'aspect et la sensation : lorsque vous échangez avec vos amis et collègues de travail au cours du processus normal d'apprentissage par les pairs, et également au cours des évaluations par les pairs, vous n'avez pas l'impression d'être en train d'apprendre. processus normal d'apprentissage par les pairs, ainsi que pendant les évaluations par les pairs, vous ne voulez pas passer du temps à décrypter leur code, mais plutôt à parler directement de ce qu'ils ont fait. pas passer du temps à décrypter leur code, mais parler directement de la logique du morceau de code.
  La norme vous demande d'utiliser une présentation spécifique, en fournissant des instructions pour la dénomination des fonctions et des variables, l'indentation, les règles d'accolade, les tabulations et les espaces à de nombreux endroits... Cela vous permettra de jeter un coup d'œil sur les codes d'autres personnes qui vous sembleront familiers, et d'aller directement à l'essentiel au lieu de passer du temps à lire le code avant de le comprendre. La norme est également une marque de fabrique. En tant que membre de la communauté 42, vous serez en mesure de reconnaître le code écrit par un autre étudiant ou ancien étudiant de 42 lorsque vous serez sur le marché du travail.

- Vision à long terme : faire l'effort d'écrire un code compréhensible est le meilleur moyen de le maintenir. Chaque fois que quelqu'un d'autre, y compris vous, doit corriger un bogue ou ajouter une nouvelle fonctionnalité, il n'aura pas à perdre

son temps précieux à essayer de comprendre ce que fait le code si, auparavant, vous aviez fait les choses de la bonne manière. Cela évitera les situations où des morceaux de code cessent d'être maintenus simplement parce que cela prend du temps, et cela peut faire la différence lorsqu'il s'agit d'avoir un produit réussi sur le marché. Le plus tôt vous apprendrez à le faire, le mieux ce sera.

- Références : vous pouvez penser que certaines, voire toutes les règles incluses dans la norme sont arbitraires, mais nous avons réellement réfléchi et lu ce qu'il fallait faire et comment le faire. Nous vous encourageons vivement à chercher sur Google pourquoi les fonctions doivent être courtes et ne faire qu'une chose, pourquoi le nom des variables doit avoir un sens, pourquoi les lignes ne doivent pas dépasser 80 colonnes, pourquoi une fonction ne doit pas prendre beaucoup de paramètres, pourquoi les commentaires doivent être utiles, etc, etc, etc...

Chapitre III

La Norme

III.1 Conventions de dénomination

- Un nom de structure doit commencer par `s_`.
- Un nom de typedef doit commencer par `t_`.
- Un nom d'union doit commencer par `u_`.
- Un nom d'enum doit commencer par `e_`.
- Un nom de globale doit commencer par `g_`.
- Les noms de variables, de fonctions doivent être composés exclusivement de minuscules, de chiffres et de '_' (Unix Case).
- Les noms de fichiers et de répertoires doivent être composés exclusivement de minuscules, de chiffres et de '_' (Unix Case).
- Les caractères ne faisant pas partie de la table ASCII standard ne sont pas autorisés.
- Les variables, fonctions, et tout autre identifiant doivent être en Snake Case. (En minuscules et en les séparant par des underscore)
- Tous les identifiants (fonctions, macros, types, variables, etc) doivent être en anglais.
- Les objets (variables, fonctions, macros, types, fichiers ou répertoires) doivent avoir les noms les plus explicites ou mnémoniques possibles.
- Les variables globales sont interdites, sauf quand vous êtes obligé d'en utiliser (signal handling). L'utilisation d'une variable globale dans un projet où ce n'est pas explicitement autorisé est une erreur de Norme.
- Le fichier doit être compilable. Un fichier qui ne compile pas n'est pas censé passer La Norme.

III.2 Formatage

- Vous devez indenter votre code avec des tabulations de la taille de 4 espaces. Ce n'est pas équivalent à 4 espaces, ce sont bien des tabulations.
- Chaque fonction doit faire au maximum 25 lignes sans compter les accolades du bloc de la fonction.
- Chaque ligne ne peut pas faire plus de 80 colonnes, commentaires compris. Une tabulation ne compte pas pour une colonne, mais bien pour les `n` espaces qu'elle représente.
- Chaque fonction doit être séparée par une ligne vide de la suivante. Tout commentaire ou préprocesseur peut se trouver juste au-dessus de la fonction. Le saut de ligne se trouve après la fonction précédente.
- Une seule instruction par ligne
- Une ligne vide doit être vide. Elle ne doit pas contenir d'espace ou de tabulation.
- Une ligne ne doit jamais se terminer par des espaces ou des tabulations.
- Vous ne pouvez pas avoir 2 espaces à la suite.
- Quand vous rencontrez une accolade, ouvrante ou fermante, ou une fin de structure de contrôle, vous devez retourner à la ligne.
- Chaque virgule ou point-virgule doit être suivi d'un espace, sauf en fin de ligne.
- Chaque opérateur et opérande doivent être séparés par un seul espace.
- Chaque mot-clé en C doit être suivi d'un espace, sauf pour ceux de type (comme `int`, `char`, `float`, etc.) ainsi que `sizeof`.
- Chaque déclaration de variable doit être indentée sur la même colonne.
- Les étoiles des pointeurs doivent être collées au nom de la variable.
- Une seule déclaration de variable par ligne
- On ne peut faire une déclaration et une initialisation sur une même ligne, à l'exception des variables globales (quand elles sont permises) et des variables statiques.
- Les déclarations doivent être en début de fonction et doivent être séparées de l'implémentation par une ligne vide.
- Aucune ligne vide ne doit être présente au milieu des déclarations ou de l'implémentation.
- La multiple assignation est interdite.
- Vous pouvez retourner à la ligne lors d'une même instruction ou structure de contrôle, mais vous devez rajouter une indentation par accolade ou opérateur d'affectation. Les opérateurs doivent être en début de ligne.

Exemple :

```
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 Fonctions

- Une fonction prend au maximum 4 paramètres nommés.
- Une fonction qui ne prend pas d'argument doit explicitement être prototypée avec le mot `void` comme argument.
- Les paramètres des prototypes de fonctions doivent être nommés.
- Chaque définition de fonction doit être séparée par une ligne vide de la suivante.
- Vous ne pouvez déclarer que 5 variables par bloc au maximum.
- Le retour d'une fonction doit se faire entre parenthèses.
- Chaque fonction doit avoir une seule tabulation entre son type de retour et son nom.

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

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

III.4 Typedef, struct, enum et union

- Vous devez mettre une tabulation lorsque vous déclarez une `struct`, `enum` ou `union`.
- Lors de la déclaration d'une variable de type `struct`, `enum` ou `union`, vous nemettrez qu'un espace dans le type.
- Lorsque vous déclarez une `struct`, `union` ou `enum` avec un `typedef`, toutes les règles s'appliquent et vous devez aligner le nom du `typedef` avec le nom de la `struct`, `union` ou `enum`.
- Vous devez indenter tous les noms de structures sur la même colonne.
- Vous ne pouvez pas déclarer une structure dans un fichier .c.

III.5 Headers

- Seuls les inclusions de headers (système ou non), les déclarations, les `defines`, les prototypes et les macros sont autorisés dans les fichiers headers.
- Tous les `includes` doivent se faire au début du fichier.
- Vous ne pouvez pas inclure de fichier C.
- On protègera les headers contre la double inclusion. Si le fichier est `ft_foo.h`, la macro témoin est `FT_FOO_H`.
- Une inclusion de header (.h) dont on ne se sert pas est interdite.
- Toute inclusion de header doit être justifiée autant dans un `.c` que dans un `.h`.

```
#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 L'en-tête 42 - c'est-à-dire commencer un fichier avec style

- Tout fichier .c et .h doit immédiatement commencer par l'en-tête standard 42 : un commentaire de plusieurs lignes avec un format spécial comprenant des informations utiles. L'en-tête standard est naturellement disponible sur les ordinateurs dans les clusters des différents éditeurs de texte (emacs : en utilisant `C-c C-h`, vim en utilisant `:Stdheader` ou `F1`, etc...)

- L'en-tête 42 doit contenir plusieurs informations à jour, notamment le créateur avec son login et son email, la date de création, le login et la date de la dernière mise à jour. Chaque fois que le fichier est enregistré sur le disque, les informations doivent être automatiquement mises à jour.

III.7 Macros et Préprocesseur

- Les constantes de préprocesseur (or `#define`) que vous créez ne doivent être utilisés que pour associer des valeurs littérales et constantes, et rien d'autre.
- Les `#define` érigés dans le but de contourner la norme et/ou obfusquer du code interdit par la norme sont interdites. Ce point doit être vérifiable par un humain.
- Vous pouvez utiliser les macros présentes dans les bibliothèques standards, si cette dernière est autorisée dans le projet ciblé.
- Les macros multilignes sont interdites.
- Seuls les noms de macros sont en majuscules.
- Il faut indenter les caractères qui suivent un `#if`, `#ifdef` ou `#ifndef`.

III.8 Choses Interdites !

- Vous n'avez pas le droit d'utiliser :
  - `for`
  - `do...while`
  - `switch`
  - `case`
  - `goto`
- Les opérateurs ternaires, comme `?`.
- Les tableaux à taille variable (VLA - Variable Length Array).
- Les types implicites dans les déclarations de variable.

```
int main(int argc, char **argv)
{
    int     i;
    char    string[argc]; // Tableau a taille variable (VLA)

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

III.9 Commentaires

- Il ne doit pas y avoir de commentaires dans le corps des fonctions. Les commentaires doivent se trouver à la fin d'une ligne ou sur leur propre ligne.
- Vos commentaires doivent être en anglais et utiles.
- Les commentaires ne peuvent pas justifier une fonction bâtarde.

III.10 Les fichiers

- Vous ne pouvez pas inclure un `.c`.
- Vous ne pouvez pas avoir plus de 5 définitions de fonctions dans un `.c`.

III.11 Makefile

Les Makefile ne sont pas vérifiés pas La Norminette. Ils doivent être vérifiés par un humain pendant l'évaluation.

- Les règles `$(NAME)`, `clean`, `fclean`, `re` et `all` sont obligatoires.
- Le projet est considéré comme non fonctionnel si le Makefile "relink".
- Dans le cas d'un projet multibinaire, en plus des règles précédentes, vous devez avoir une règle `all` compilant les deux binaires ainsi qu'une règle spécifique à chaque binaire compilé.
- Dans le cas d'un projet faisant appel à une bibliothèque de fonctions (par exemple une `libft`), votre makefile doit compiler automatiquement cette bibliothèque.
- Les sources nécessaires à la compilation de votre programme doivent être explicitement citées dans votre Makefile.