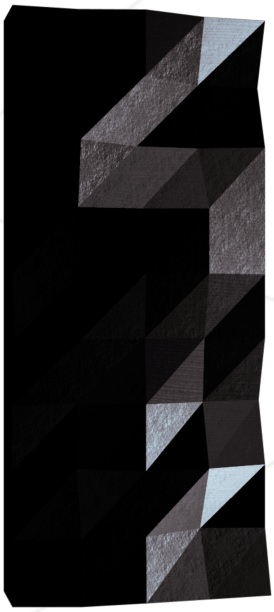

La norme de 42

Version 1.8

Benny benny@42.fr
Thor thor@42.fr
marvin marvin@42.fr

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. Il est obligatoire de respecter la norme lorsque vous écrivez du C à 42.

Table des matières

Chapitre I

Avant-propos

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. Il est obligatoire de respecter la norme lorsque vous écrivez du C à 42.

I.1 Pourquoi imposer une norme ?

La norme a deux objectifs principaux :

- Uniformiser vos codes afin que tout le monde puisse les lire facilement, étudiants et encadrants.
- Ecrire des codes simples et clairs.

I.2 La norme dans vos rendus

Tous vos fichiers de code C doivent respecter la norme de 42. La norme sera verifiée par vos correcteurs et la moindre faute de norme donnera la note de 0 à votre projet ou à votre exercice.

La moulinette de correction utilisée en plus de vos soutenances lancera un programme appellé "Norminette". La Norminette vérifiera le sous-ensemble des règles de norme qu'il lui est possible de vérifier.

Lors de vos peer correctings, vous ne devez verifier que les règles de norme que la Norminette est capable de vérifier dans sa version actuelle. Ainsi tout le monde sera égal face à la norme. Vous trouverez la liste des règles de norme que la Norminette gère à un instant t à la fin de ce document. Cette liste sera régulièrement mise à jour par le bocal, gardez donc un oeil dessus.

I.3 Conseils

Comme vous le comprendrez rapidement, la norme n'est pas une contrainte. Au contraire, la norme est un garde-fou pour vous guider dans l'écriture d'un C simple et basique. C'est pourquoi il est absolument vital que vous codiez directement à la norme, quite à coder plus lentement les premières heures. Un fichier de sources qui contient une faute de norme est aussi mauvais qu'un fichier qui en compte dix. Soyez studieux et appliqués, et la norme deviendra un automatisme sous peu.

Chapitre II

La norme de 42

II.1 Convention de dénomination

- Les objets syntaxiques (variables, fonctions, macros, types, fichiers ou répertoires) doivent avoir les noms les plus explicites ou mnémoniques. Seul les “compteurs” peuvent être nommés à votre guise.
- Les abréviations sont tolérées dans la mesure où elles permettent de réduire significativement la taille du nom sans en perdre le sens.
- Les parties des noms composites seront séparées par ’_’ (underscore).
- Tous les identifiants (fonctions, macros, types, variables, etc.) doivent être en anglais. Pas de français ou de “franglais”.
- 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_".
- Toute utilisation de variable globale doit être justifiée.
- Les noms de variables, de fonctions, de fichiers et de répertoires doivent être composés exclusivement de minuscules, de chiffres et de ’_’. (Unix Case)

```
void    ft_do_stuff(void);      /* WRONG : Nous ne savons pas ce que fait cette fonction */

void    ft_print_char(void);    /* RIGHT : Nous savons que cette fonction affiche un char */

void    ft_testing_function(void)
{
    int     bla;                /* WRONG */
    int     prout;              /* WRONG */
    char    *gruik;             /* WRONG */
}

void    ft_testing_function(void)
{
    int     counter;            /* RIGHT */
    int     user_input_size;    /* RIGHT */
    char    *user_input;        /* RIGHT */
}

int ft_calculator_function(void);   /* RIGHT : Unix case */
int ftCalculatorFunction(void);     /* WRONG : Caml case */
int FtCalculatorFunction(void);     /* WRONG : Java case */
```

II.2 Formatage

- Tous vos fichiers devront commencer par le header standard de 42 dès la première ligne. Le header ressemble à :

```
/******************************************************************************/
/*                                                                            */
/*                                                        :::      ::::::::   */
/*   filename_____________________________.ext          :+:      :+:    :+:   */
/*                                                    +:+ +:+         +:+     */
/*   by: login____ <login____@student.42.fr>        +#+  +:+       +#+        */
/*                                                +#+#+#+#+#+   +#+           */
/*   Created: yyyy/mm/dd hh:mm:ss by login____          #+#    #+#             */
/*   Updated: yyyy/mm/dd hh:mm:ss by login____        ###   ########.fr       */
/*                                                                            */
/******************************************************************************/
```

- Chaque fonction doit faire au maximum 25 lignes sans compter les accolades du bloc de la fonction. Exemple d'une fonction de 5 lignes :

```
int ft_fct_demo(void)
{
    int count;

    count = 41;
    count = count + 1;
    return (count);
}
```

- Vous devez indenter votre code avec des tabulations de 4 espaces (Ce n'est pas équivalent à 4 espaces, ce sont bien des tabulations.). Dans sa configuration de base, votre éditeur est susceptible d'insérer des espaces en lieu et place de tabulations, soyez attentifs. Consultez la documentation de votre éditeur si vous avez un doute.
- Chaque ligne ne peut faire plus de 80 colonnes, commentaires compris. (Attention, une tabulation ne compte pas pour une colonne mais bien pour les n espaces qu'elle représente).

- Une seule déclaration par ligne.
- Une seule instruction par ligne.
- Une ligne vide ne doit pas contenir d'espaces ou de tabulations.
- Une ligne ne doit jamais se terminer par des espaces ou des tabulations.
- Une accolade ouvrante ou fermante doit être seule sur sa ligne avec la bonne identation. Le cas particulier des typedef de struct/union/enum est traité plus bas.
- Vous devez retourner à la ligne à la fin d'une structure de contrôle (if, while, etc.).
- Vous devez mettre des accolades après une structure de contrôle, si le bloc comporte plus d'une instruction.

Exemples :

```
if (test > 0 && test < 42) { return (value); }    /* WRONG */

if (test > 0 && test < 42) return (value);        /* WRONG */

if (test > 0 && test < 42)
{
return (value);                                   /* WRONG */
}

if (test > 0 && test < 42)
{
        return (value);                           /* WRONG */
}

if (test > 0 && test < 42) {
    return (value);                               /* WRONG */
}

if (a = 42)                                       /* RIGHT */
    a = 0;

if (a = 42)                                       /* RIGHT */
{
    a = 0;
}

if (a = 42)                                       /* WRONG */
    if (b = 42)
        a = 0;

if (a = 42)                                       /* RIGHT */
{
    if (b = 42)
        a = 0;
}
```

- Chaque virgule ou point-virgule doit être suivi d'un espace si nous ne sommes pas en fin de ligne.
- Chaque opérateur (binaire ou ternaire) et ses opérandes doivent être séparés par un espace et seulement un. Toutefois il ne doit pas y avoir d'espace entre un operateur unaire et son opérande.
- Chaque mot clef du C doit être suivi d'un espace sauf pour les spécifieurs de type (comme int, char, float, const, etc.) ainsi que sizeof.

- L'expression retournée avec le mot clef "return" doit être entre parenthèses.

Exemples :

```
if(test==0)
{
    return (value);           /* WRONG */
}

if (test==0)
{
    return (value);           /* WRONG */
}

if(test == 0)
{
    return (value);           /* WRONG */
}

if (test == 0)
{
    return value;             /* WRONG */
}

if (test == 0)
{
    return (value);           /* RIGHT */
}

if (test == 0)
{
    return ;                  /* RIGHT */
}
```

- Chaque déclaration de variable doit être indentée sur la même colonne. Les étoiles des pointeurs doivent être collées au nom de la variable et les unes aux autres.
- Le type d'une variable et son identifiant doivent être séparés par au moins une tabulation.
- Une seule déclaration de variable par ligne.
- On ne peut PAS faire une déclaration et une initialisation sur une même ligne, à l'exception des variables globales, des variables statiques, des constantes et des initialisations de tableaux. Dans ce dernier cas en particulier, l'expression d'initialisation du tableau doit être à la norme (accolades, virgules, ...).

Exemples :

```
void    main(void)
{
    char letter;                /* WRONG */
    double current_value;
    char *words;

    letter = 'c';
}

void    main(void)
{
    char    letter = 'c';       /* WRONG */
    double  current_value;
    char    *words;
}

void    main(void)
{
    char    letter;             /* RIGHT */
    double  current_value;
    char    *words;

    letter = 'c';
}
```

- Les déclarations doivent être en début de bloc 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.

```
void    main(void)
{
    char    letter;         /* WRONG */
    double  big_number;
    char    *words;
    letter = 'a';
    big_number = 0.2;
}

void    main(void)
{
    char    letter;         /* WRONG */
    double  big_number;
    char    *words;

    letter = 'a';

    big_number = 0.2;
}

void    main(void)
{
    char    letter;         /* RIGHT */
    double  big_number;
    char    *words;

    letter = 'a';
    big_number = 0.2;
}
```

- Vous pouvez retourner à la ligne lors d'une même instruction ou structure de contrôle, mais vous devez rajouter une indentation par parenthèse ou opérateur

d'affectation. Les opérateurs doivent être en début de ligne. Retourner à la ligne nuit à la lisibilité de votre code, soyez donc mesurés. Plus généralement, si vous avez des instructions ou des expressions trop longues, c'est que vous n'avez pas suffisamment factorisé votre code.

```
toto = 42 + 5           /* RIGHT (Mais bof) */
    - 28;

if (toto == 0
    && titi == 2
    && (tutu == 3       /* RIGHT (Mais bof) */
        && tata == 4)
    || rara == 3)
```

II.3 Paramètres de fonction

- 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.

```
int fct();              /* WRONG */
int fct(void);          /* RIGHT */

int main(void)
{
  fct();                /* RIGHT */
  fct(void);            /* WRONG */
  return (0);
}
```

II.4 Fonctions

- Les paramètres d'un prototype de fonction doivent posséder des noms.
- Vous ne pouvez déclarer que 5 variables par bloc au maximum.
- Vos identifiants de fonction doivent être alignés dans un même fichier (s'applique aux headers).
- Chaque définition de fonction doit être séparée par une ligne vide.
- Le type de retour d'une fonction et l'identifiant de cette fonction doivent être séparés par au moins une tabulation.

```
void        main(void)
{
        ...
}
                                                /* WRONG */
struct s_info   get_info(void)
{
        ...
}

/*=============================================*/

void            main(void)
{
        ...
}
                                                /* RIGHT */
struct s_info   get_info(void)
{
        ...
}
```

- Souvenez vous que l'on met toujours un espace après une virgule ou un point-virgule si nous ne sommes pas en fin de ligne.

```
void    fct(int arg1, int arg2, int arg3)
{
    ...
}

int     main(void)
{
    ...

    fct(arg1, arg2, arg3);

    ...
}
```

II.5 Typedef, struct, enum et union

- Les structs, unions et enums anonymes sont interdites.
- Les déclarations de structs, unions et enums doivent être dans le scope global.
- Vous devez mettre une tabulation avant l'identifiant lorsque vous déclarez une struct, enum ou union.

```
struct s_buff
{
        ...                                     /* WRONG */
};

struct  s_buff
{
        ...                                     /* RIGHT */
};
```

- Lors de la déclaration d'une variable de type struct, enum ou union vous ne mettrez qu'un espace dans le type.

```
struct  s_buff  toto;                           /* WRONG */
struct s_buff   toto;                           /* RIGHT */
```

- Vous devez utiliser une tabulation entre les deux paramètres d'un typedef.

```
typedef int myint;      /* WRONG */
typedef int     myint;  /* RIGHT */
```

- 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.
- Dans ce cas particulier, le nom du typedef pourra être sur la même ligne que l'accolade fermante.

```
typedef struct s_buff
{
        ....                                    /* WRONG */
}           t_buff;

typedef struct  s_buff
{
        ....                                    /* WRONG */
}t_buff;


typedef struct  s_buff
{
        ....                                    /* WRONG */
}   t_buff;

typedef struct  s_buff
{
        ....                                    /* RIGHT */
}               t_buff;
```

II.6 Headers

- Seuls les inclusions de headers (système ou non), les définitions de structures de données, les defines, les prototypes et les macros sont autorisés dans les fichiers headers.
- Toute inclusion de header doit etre justifiée autant dans un .c que dans un .h.
- Tous les includes de .h doivent se faire au début du fichier (.c ou .h).
- La norme s'applique aussi aux headers.
- Les headers doivent être protégés contre la double inclusion. Si le fichier est foo.h, la macro témoin est FOO_H.

```
#ifndef FOO_H
# define FOO_H

/* code du header */

#endif /* !FOO_H */
```

- Dans le cas unique et précis de la ligne de commentaire à droite d'un `#endif`, la norme des commentaires ne s'applique pas (voir plus bas dans ce document).
- Une inclusion de header (.h) dont on ne se sert pas est interdite.
- Les macros doivent se trouver exclusivement dans des fichiers .h. Les seules macros tolérées dans les fichiers .c sont celles qui activent des fonctionnalités (ex : BSD_SOURCE).

II.7 Macros et Pré-processeur

- Les macros multilignes sont interdites.
- Seuls les noms de macros sont en majuscules.

```
#define FOO ‘‘bar’’
```

- Il faut indenter les caractères qui suivent un #if , #ifdef ou #ifndef avec un espace à chaque niveau.

```
#ifndef TOTO_H
# define TOTO_H
# ifdef __WIN32
#  define FOO "bar"
# endif /* __WIN32 */
#endif  /* !TOTO_H */
```

- Pas de #if, #ifdef ou #ifndef après la première définition de fonction dans un .c.

II.8 Choses Interdites !

Vous n'avez pas le droit d'utiliser :

- for
- do ...while
- switch
- case
- goto
- Les ternaires imbriqués

```
(a ? (b ? : -1 : 0) : 1)  /* WRONG */
```

- Les ternaires servant à autre chose qu'à une affectation.

```
(a == 42 ? fun1() : fun2());      /* WRONG */
a = (a == 42 ? fun1() : fun2());  /* RIGHT */
```

II.9 Commentaires

- Les commentaires peuvent se trouver dans tous les fichiers source.
- Il ne doit pas y avoir de commentaires dans le corps des fonctions.
- Les commentaires sont commencés et terminés sur une ligne seule. Toutes les lignes intermédiaires s'alignent sur elles, et commencent par "**".
- Pas de commentaire C++ ("//").
- Vos commentaires doivent être en anglais et utiles.
- Le commentaire ne peut pas justifier une fonction tordue.

```
/*
** This function makes people laugh
*/
void    ft_lol(void)
{
}

/*
** Right
*/

/*
 * Wrong
*/
```

II.10 Les fichiers

- Vous ne pouvez pas inclure un .c. Jamais. Même si quelqu'un vous a dit de le faire.
- Vous ne pouvez pas avoir plus de 5 définitions de fonction dans un .c.

II.11 Makefile

- 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 tous les 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.
- L'utilisation de wildcard (*.c par exemple) est interdite.

Chapitre III

La Norminette

Dans sa version actuelle, la Norminette vérifie les règles présentes dans cette section. Nous mettrons cette section à jour régulièrement. Bien entendu, toutes les règles de norme sont à respecter, y compris celles que la Norminette ne vérifie pas encore et celles qui ne sont pas vérifiables automatiquement, mais vous ne devez vérifier que les règles suivantes pendant les peer correcting. Référez-vous aux sections précédentes pour avoir le détail de chaque règle.

- Tous vos fichiers devront commencer par le header standard de 42 dès la première ligne.
- Chaque fonction doit faire au maximum 25 lignes sans compter les accolades du block de la function.
- Chaque ligne ne peut faire plus de 80 colonnes, commentaire compris. (Attention une tabulation ne compte pas pour 1 colonne mais bien pour les n espaces qu'elle représente).
- Une seule instruction par ligne.
- Une ligne vide ne doit pas contenir d'espaces ou de tabulations.
- Une ligne ne doit jamais se terminer par des espaces ou des tabulations.
- Quand vous avez une fin de structure de contrôle, vous devez retourner à la ligne.
- Vous devez mettre un espace après un mot clef.
- Vous devez mettre un espace après une virgule ou un point virgule si ce n'est pas la fin de la ligne.
- Vous devez mettre un espace avant et après un opérateur binaire ou ternaire.
- Vous n'avez pas le droit d'utiliser : for, do . . .while, switch, case et goto.
- Les commentaires C++ ("//") sont interdits.
- Vous ne pouvez faire que 5 définitions de fonction par fichier.
- Une fonction ne peut prendre que 4 paramètres.