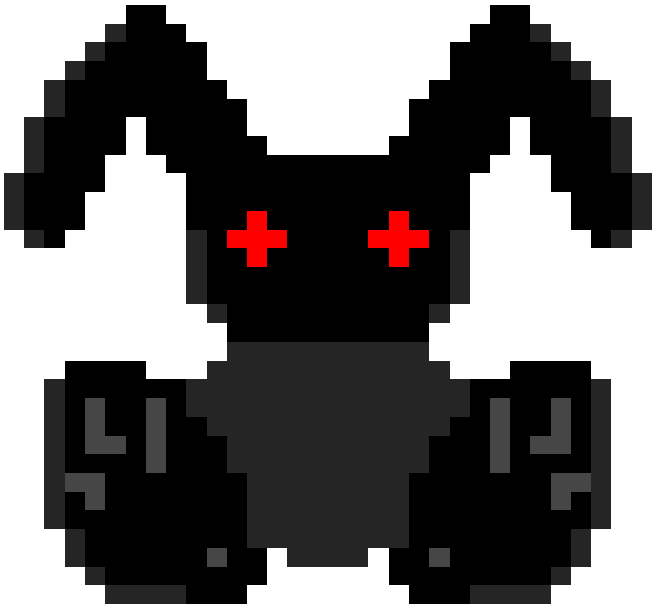
D A E M O N L A B



LAPINS NOIRS

Tables de la Norme

La Vérité sous la forme d'un style

- DaemonLab -

- La Caverne aux Lapins Noirs -

Ce document présente le style à respecter impérativement lors de la rédaction de code en langage C et de Makefile, à moins qu'une autre norme ne soit indiqué.

INDEX

01 – Avant-propos

Les Tables de la Norme sont un ensemble de règle à respecter impérativement pour toutes les activités en langage C et pour tous les Makefile UNIX, sauf exception notifié dans les activités correspondantes.

Les objectifs des Tables de la Norme sont d'uniformiser vos productions mais également d'introduire par la contrainte certains réflexes requis par notre enseignement, tel que le découpage en fonction.

Soyons clairs : le fait que ces règles soient immuables et impératives en toute circonstance est une **stupidité**. La programmation implique des cas particuliers où ces règles seront plus une difficulté qu'un avantage, néanmoins, **vous les respecterez tout de même**. En effet : un programme de vérification de norme passera sur vos productions et ne saurait juger de la pertinence de vos exceptions.

Différents sujets d'activités lèveront certaines interdictions afin de mettre en pratique certains aspects du langage C ou des Makefile UNIX qui seraient prohibés par les tables de norme dans le cas contraire.

Chaque règle est accompagné du nombre de « points d'erreurs » que provoque le non-respect de la règle. Le nombre de point d'erreur s'accumule pour un type d'erreur jusqu'à trois fois. Les conséquences de l'accumulation du nombre de point d'erreur est détaillé à la fin de ce document.

02 – Règles générales

Tous les fichiers doivent commencer par l'en-tête réglementaire de l'école. Cette en-tête est accessible via un raccourci pré-configuré dans votre **emacs** : CTRL-C H.

```
/*
** Prénom Nom Pseudonyme
** Login - <login@domain>
**
** NomDuProjet - AnnéeDuProjet
*/
```

L'oubli d'un en-tête apporte 5 points d'erreur.

L'indentation doit être réalisé par les smart-tab pré-configuré d'**emacs**. Cette indentation est appelée via la touche tabulation. Les tabulations forcées doivent être réalisé via le raccourci de **emacs** : ALT+i.

Une indentation mal réalisée apporte 1 points d'erreur.

Il ne peut y avoir qu'une seule instruction ou déclaration par ligne.
Les variables ne peuvent pas être assignées sur la ligne de leur déclaration, sauf si il s'agit d'une static et que la valeur assignée n'est pas celle par défaut.

Bon

```
int i;
int j;

i = 1;
j = i + 2;
```

Mauvais

```
int i, j;
int k = 0;

i = 1, j = 2;
```

Enfreindre cette règle apporte 3 point d'erreur.

Aucune ligne ne doit excéder 80 colonnes.

Enfreindre cette règle apporte 1 point d'erreur par ligne excessive.

On placera un espace autour chaque opérateur binaire, à l'exception de la virgule.
On placera un unique espace après la virgule.
Les parenthèses n'induisent pas d'espace.
Les opérateurs unaires n'induisent pas d'espace.

Bon

```
i = 5 + 6 * 3++;
j = (4 + 5) * 47;

sum(i, j);
```

Mauvais

```
i=5+7;
j = ( 4+5 )* 50;

sum(i,j);
```

Enfreindre cette règle apporte 1 point d'erreur par ligne comportant une erreur.

Tous les fichiers doivent terminer par une ligne vide. C'est à dire que les deux derniers caractères d'un fichier doivent être « \n\n ».

Enfreindre cette règle apporte 1 point d'erreur par fichier.

Le format des fichiers textes sera celui d'UNIX.

Enfreindre cette règle apporte 1 point d'erreur par fichier.

Aucun espace, aucune tabulation ne doit être situé en fin de ligne.
Une fonction de **emacs** peut vous aider à vous en débarrasser. Invoquez la ligne de commande à l'aide de ALT x et tapez **delete-trailing-whitespace**.

Enfreindre cette règle apporte 1 point d'erreur par ligne comportant une erreur.

03 – Fichier code

Une fonction ne doit pas excéder 25 lignes entre ses accolades.
L'opérateur virgule est interdit. La virgule n'est autorisée que dans les appels de fonction et comme séparateur de paramètres.
Le symbole du pointeur « * » doit être situé du coté du symbole et non du coté du type.
Les structures de contrôles sont toujours suivi d'un saut de ligne immédiat.
Tous les variables doivent être déclarés au début de la fonction.
Il doit y avoir une ligne vide entre les déclarations de variable et le reste de la fonction.
La fonction ne doit comporter aucune autre ligne vide.

Bon

```
void   *func(int      *param)
{
  int  *i;
  int  j;

  if (param != NULL)
    i = param;
  j = 7;
}
```

Mauvais

```
void*   func(int*   param)
{
  int*  i;
  if (param != NULL) i=param;
  int j;
  j = 7, *i += 2;
}
```

Une fonction trop longue, une virgule ou l'absence de saut de ligne après une structure de contrôle apporte 5 points d'erreur. Un autre type d'erreur apporte 1 point d'erreur.

Les accolades dans les fonctions et les globales doivent être pré-indenté.
Les accolades doivent être seules sur leur ligne, sauf celle fermant un **typedef**, terminant par un point-virgule ou par une autre accolade fermante, type « {} ». Si il est possible de se passer des accolades, il faut s'en passer.

Bon

```
if (condition)
  {
     j = 2;
     i = 4;
  }
if (condition)
  k = 8;
```

Mauvais

```
if (condition) {
  j = 2;
}
if (condition)
{
  i = 4;
  k = 8;
}
```

Une erreur de ce type apporte 1 point d'erreur.

Il est interdit de placer des commentaires dans les fonctions. Les commentaires concernant une fonction doivent être situés au dessus de celle-ci. Si le commentaire fait plusieurs lignes, alors il est obligatoire d'utiliser le format de commentaire multi-ligne !

Les commentaires multi-lignes commencent par /* situés seuls sur une ligne et terminent par */ situés également seul sur une ligne. Toutes les lignes intermédiaires commencent par **.
Si un générateur de documentation est utilisé, la première ligne peut-être modifié comme c'est requis par le dit générateur.

Les commentaires peuvent être en français ou en anglais, si une obligation n'est pas indiqué dans le sujet de l'activité.

Bon

```
/*
** Ceci est un commentaire
** autorisé
*/
void func(void)
{}

// Autorisé car court
void fund(void)
{}
```

Mauvais

```
/* mauvais car pas // */
/*
 * mauvais
*/
/*
   mauvais
 */

// C'est aussi mauvais car
// multi-ligne avec //
```

Une erreur de ce type apporte 1 point d'erreur.

Les constantes magiques sont interdites. Une constante magique est une constante non symbolique, c'est à dire une valeur écrite « en dur » dans le code. Les seules constantes magiques autorisées sont celles qui sont traditionnellement employé et reconnaissable : 0 et -1.

Bon

```
write(STDOUT_FILENO, s, strlen(s));

i = MAGIC_TAR_FILE;

c = 'e';
```

Mauvais

```
write(1, s, 1);

i = 0xFE;

c = 101;
```

Une erreur de ce type apporte 2 point d'erreur.

Une fonction ne peut pas prendre plus de 4 paramètres.
Les structures et unions de plus de 16 octets doivent être passé par adresse.
Un paramètre passé par adresse non modifié dans la fonction doit être constant.
Il doit y avoir un seul paramètre par ligne.
Les noms des paramètres doivent être alignés.
Les noms des variables doivent être alignés avec le nom de la fonction.
L'ensemble des noms de fonction d'un fichier doivent être aligné.
Les noms des globales doivent être alignés avec les noms de fonction.
Les symboles de précompilation doivent être alignés avec les noms de fonctions.
Les définition de symboles de précompilation doivent être alignés avec les paramètres.

Bon

```
#include <stdio.h>
#define  SYMBOL                value
void     f(t_big               *s,
           const char          *str)
{
  int    i;

  s->x = 7;
  puts(str);
}
```

Mauvais

```
#include <stdio.h>
#define SYMBOL value
void g(t_big s, char *s, int a, int b, int c)
{
  int     i;
  printf("%d", s->x);
  puts(str);
}
```

Un paramètre en trop apporte 3 points d'erreur. Une erreur de mise en page 1 point.

Un espace doit être placé à la suite du mot-clef d'une structure de contrôle et la parenthèse qui lui est associé. Des parenthèses doivent être associés au mot clef return.

Bon

```
if (condition)
{}
while (condition)
{}
for (condition)
{}
switch (condition)
{}
return (value);

break ;
continue ;
goto abc;
```

Mauvais

```
if(condition)
{}
while(condition)
{}
for(condition)
{}
switch(condition)
{}
return(value);
return value;
break;
continue;
goto abc;
```

Une erreur de ce type apporte 1 points d'erreur.

Les mot-clefs **goto**, **break**, **for** et **switch** sont interdits. De nombreuses activités peuvent vous autoriser un ou plusieurs de ces mot-clefs au cas par cas.

L'utilisation de **for** ou **goto** apporte 5 points d'erreur. Une autre mot clef 2 points d'erreur.

Les variables globales non constantes sont interdites.
Les variables globales constantes sont à justifier.

L'utilisation d'une globale non constante apporte 5 points d'erreur.

Toute inclusion de fichier en-tête doit être motivé. Un fichier en-tête inclus sans raison est une faute.

Une erreur de ce type apporte 1 points d'erreur.

04 – Nommage et en-êtes

Les symboles doivent être en anglais. Ils doivent avoir des noms explicites. Si des mnémoniques sont utilisés, celles-ci doivent être évidentes. Les index des boucles peuvent être i, j et k. Les noms composés doivent être séparés par des '_'.

Les symboles de fonction, de variable et de type doivent être en minuscules.
Les constantes symboliques et macros du préprocesseur doivent être en majuscule.
Les valeurs énumérés doivent être en majuscule.

Un type structure doit commencer par « s_ ».
Un type énumération doit commencer par « e_ ».
Un type union doit commencer par « u_ ».
Un type défini via un typedef doit commencer par « t_ ».
Un typedef sur un type « s_abc », « e_abc », « u_abc » doit s'appeler « t_abc ».
Un symbole de globale doit commencer par « g_ ».

Le nom du typedef doit être aligné avec le nom de la structure.
Le nom des attributs d'un bloc doivent être aligné avec le nom du bloc.
Les noms des structures, unions et énumérations doivent être alignés sur le fichier.
Les noms de fonctions, paramètres de fonction, macros doivent suivre l'alignement.

Bon

```
#define         DEFAULT_FRACTAL     MANDEL

typedef enum    e_type
{
  CARPET,
  MANDEL
}               t_type;

typedef struct s_fractal
{
  int           flow_of_sharpness;
  t_type        type_of_fractal;
}               t_fractal;

void            draw(t_fractal      *ftl);
```

Mauvais

```
#define DEFAULT_FRACTAL MANDEL

typedef enum e_type
{
  CARPET,
  MANDEL
} t_type;

struct s_fractal
{
  int           flow_of_sharpness;
  t_type        type_of_fractal;
};
typedef struct s_fractal t_fractal;
void draw(t_fractal *ftl);
```

Une erreur de ce type apporte 1 points d'erreur.

Les directives de préprocesseur doivent être indenté avec un unique espace par niveau d'indentation à placer entre le symbole '#' et le mot-clef. Les mot-clefs #if, #ifdef et #endif provoquent cette indentation.

Lorsque la distance entre un #if ou un #ifdef et son #endif correspondant est grande (plus de 15 lignes), on rappellera la condition dans un commentaire en ligne situés à la suite du #endif et séparé d'un unique espace.

Tous les fichiers en-tête doivent être protégé contre la multiple inclusion à l'aide d'une macro-témoin portant le nom du fichier en majuscule. Par exemple, le fichier macro.h disposera d'une macro-témoin MACRO_H.

Bon

```
#ifndef         HEADER_H
# define        HEADER_H
# include       <lapin.h>
# define        MAGIC_INTEGER       0x42

typedef struct s_type
{
  int           integer;
}               t_type;

#endif //       HEADER_H
```

Mauvais

```
#include      <lapin.h>
#define MAGIC_INTEGER 0x42

typedef struct s_type
{
  int           integer;
}               t_type;
```

Une erreur de ce type apporte 1 points d'erreur.

Toute collection de constante de précompilation lié par une sémantique commune doivent être associé dans une énumération.

Bon

```
typedef enum    e_type
{
   INTEGER,
   DOUBLE,
   STRING
}               t_type;
```

Mauvais

```
#define INTEGER 0
#define DOUBLE 1
#define STRING 2
```

Une erreur de ce type apporte 1 points d'erreur.

Les éléments d'une structure doivent être lié par une sémantique commune. Une structure fourre tout est une faute.

Les macro multi-lignes sont interdites.

Il est interdit d'instancier une variable globale dans un fichier en-tête, il est seulement autorisé d'en faire la déclaration.

Une macro multi-ligne apporte 5 points d'erreur. Les autres un seul point.

05 – Organisation générale des fichiers

Un fichier .c ne doit pas contenir plus de 5 fonctions.
Un fichier .c ne peut contenir qu'une seule fonction n'étant pas static.
Si le projet comporte une association entre certains fichier en-tête et certains fichiers .c, les fichiers .c doivent être situés dans un dossier portant le nom du fichier en-tête.
Le fichier .c doit porter le nom exact de sa fonction non static. Ils peuvent être dépourvu de certains préfixes si celui-ci se retrouve dans l'arborescence de dossier dans lequel il se trouve. Par exemple, le fichier contenant la fonction printer_print_pattern peut s'appeler print_pattern si il est dans le dossier printer/.
Les fichiers n'ayant pas d'association particulière (tel que le fichier contenant la fonction main) doivent être situés dans le dossier src/.

Bon

```
$> ls -R
./include/graphics.h
./include/sound.h
./sound/play.c
./graphics/draw.c
./src/main.c

$> cat include/graphics.h
#ifndef     GRAPHICS_H
# define    GRAPHICS_H
void        draw(void) ;
#endif //   GRAPHICS_H

$> cat graphics/play.c
/*
**
*/
#include    "graphics.h"

static void draw_stuff(void)
{}

void        draw(void)
{
   draw_stuff();
}
```

Mauvais

```
$> ls -R
./include/project.h
./src/project.c
```

Un trop-plein de fonction dans un fichier apporte 3 points d'erreur.
Une non corrélation entre le nom d'un fichier et sa fonction 1 point d'erreur.
Une non corrélation entre le nom d'un dossier et son en-tête, 1 point d'erreur.
La présence de plus d'une fonction non static, 1 point d'erreur.

06 – Makefile

Un **Makefile** doit comporter les règles **all**, **clean**, **fclean**, **re** et **install**.

La règle **all** est la règle par défaut. La règle all compile tous les fichiers puis relie les fichiers objets en un produit final, qu'il s'agisse d'une bibliothèque ou d'un exécutable.
La règle **clean** supprime tous les fichiers objets.
La règle **fclean** appelle la règle **clean** puis supprime le produit final.
La règle **re** effectue un **fclean** suivi d'un **all**.
La règle **check** exécute les tests unitaires (et génère la couverture de code si possible).
La règle **install** procède à l'installation dans **/usr/local/**. Si le produit final est un exécutable, le dossier final est **bin/.** Si le produit final est une bibliothèque, les fichiers en-tête dans **include/** et le fichier de bibliothèque dans **lib/**.

Votre **Makefile** par défaut doit générer un produit « **release** » disposant *d'optimisations*. Si la variable **DEBUG** est mise à « y », au contraire, aucune optimisation n'est effectuée et des symboles de debug sont ajoutés.

Un projet est considéré non fonctionnel si :
- Lorsqu'on tente de recompiler, les fichiers objets étant déjà compilé sont recompilés.
- Lorsqu'on tente de relier les fichiers objets, le produit final est régénéré alors qu'il est déjà présent.

Si un programme dispose de plusieurs produits de compilation, le **Makefile** doit disposer d'une règle pour chaque.

Vous avez la possibilité d'utiliser des **sous-Makefile** tant que le fonctionnement externe reste celui qui est décrit.

Sauf exception, vos compilations doit utiliser les options **-Wall -Wextra -Werror -std=c11**. En cas d'absence de ces options, elles seront ajoutés à la correction, si votre programme ne compile plus, votre projet sera considéré non fonctionnel.

07 – Notation

Vous gagnerez une médaille de norme parfaite si vous n'avez aucun point d'erreur.

Vous gagnerez une médaille de norme si vous avez moins de 7 points d'erreur.

Si vous avez 20 ou plus points d'erreurs, votre projet sera considéré non fonctionnel.

08 – Tests

Vos tests seront placés dans le dossier **tests/src/** et le dossier **tests/** lui-même contiendra un **Makefile**. La couverture de code sera écrite dans un dossier **tests/html** généré après le lancement des test et **qui ne doit pas être rendu.**

La couverture de test sera générée par **genhtml** à la suite de l'exécution de **lcov**.