



# Norme C-Epitech

Mars 2011

Responsable Astek astek_resp@epitech.eu

# Table des matières

# Qu'est-ce que c'est ?

- La norme C-Epitech est une convention de programmation qui a été créé par l'école.
- Elle concerne :
  - La dénomination des objets
  - La présentation globale (paragraphes)
  - La présentation locale (lignes)
  - Les headers (fichiers includes)
  - Le Makefile

`Le norme est une convention purement syntaxique, à ce titre elle ne peut être utilisée comme excuse si votre programme ne fonctionne pas !`

# Quand s'applique-t-elle ?

- La norme est obligatoire dans les modules gérés par les Asteks, en voici une liste non-exhaustive :
  - B1 - Système Unix
  - B1 - C - Prog Elem
  - B1 - Igraph
  - B2 - Systeme Unix
  - B2 - C - Prog Elem
  - B2 - Igraph
  - B4 - Systeme Unix

Bien que la norme ne soit pas obligatoire dans tous les projets, ce n'est pas une raison pour ne pas toujours séquencer et structurer votre code !

*Indices* On ne remet pas à la norme ! On code à la norme ! C'est-à-dire qu'on écrit à la norme dès le début de son programme.

# Pourquoi existe-t-elle ?

- Uniformiser l'écriture des programmes au sein de l'école.
- Structurer le code.
- Faciliter la lecture du code.

`Certains choix dans la norme peuvent paraître un peu trop restrictifs, mais n'oubliez pas que la norme est aussi un outil pédagogique, les 25 lignes sont là par exemple pour accentuer le fait que vous devez absolument structurer vos programmes.`

# Présentation générale

- Une ligne ne doit pas excéder 80 colonnes.
- Une seule instruction par ligne.
- Une fonction ne doit pas excéder 25 lignes entre les accolades.

```
1   /*
2   ** cette fonction fait 2 lignes
3   */
4   int hello_world(void)
5   {
6       my_putstr("Hello world\n");
7       return (0);
8   }
```

- Un fichier ne doit pas contenir plus de 5 fonctions.
- Seuls les définitions, les defines, les prototypes et les macros sont autorisés dans les fichiers headers.
- Les macros multilignes sont interdites.
- Les fichiers source (.c, .h et Makefile par exemple) doivent toujours commencer par le header standard de l'école. Ce header est créé sous emacs à l'aide de la commande C-c C-h.

# Les commentaires

- Il ne doit pas y avoir de commentaires dans le corps des fonctions.
- Les commentaires sont commencés et terminés par une ligne seule. Toutes les lignes intermédiaires s'alignent sur elles, et commencent par '**'

```
1  /*
2  ** cette fonction calcule ....
3  */
4  void *func_mort_de_rire()
5  {
6  }
7
8  /*
9  ** Correct
10 */
11
12 /*
13  * Incorrect
14  */
```

# Indentation générale

- L'indentation sera celle obtenue avec Emacs (avec la configuration dans $HOME/../.env/.emacs). Elle se fait au moyen de la touche Tab.

```
1   int         get_type(char type_char)
2   {
3       t_types *types;
4
5       types = types_tab;
6       while (types->type_val)
7         {
8           if (types->type_val == type_char)
9               return (types->type_val);
10          types++;
11        }
12      return (FALSE);
13  }
```

# Nomination

- Les objets (variables, fonctions, macros, types, fichiers ou répertoires) doivent avoir les noms les plus explicites ou mnémoniques.
- 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 '_'.
- Tous les identifiants (fonctions, macros, types, variables etc.) doivent être en anglais.
- Les noms de variables et de fonctions de fichiers et de répertoires doivent être composés exclusivement de minuscules, de chiffres et de '_'.
- Seuls les noms de macros sont en majuscules.
- Un typedef sur s_ma_struct doit s'appeler t_ma_struct.

```
1    struct s_my_struct
2    {
3    }      t_my_struct;
```

- Un nom de structure doit commencer par s_.
- Un nom de type doit commencer par t_.
- Un nom de globale doit commencer par g_.

```
1    #define FOO "bar"
2
3    void hello_world()
4    {
5    }
```

# Déclarations / Affectations

- Les fonctions et variables non exportées par foo.c sont déclarées en static.
- On sautera une ligne entre la déclaration de variable et les instructions. On met une variable par ligne. Il ne doit pas y avoir d'autres lignes vides dans les blocs. Si vous avez envie de séparer des parties d'un bloc, faites soit plusieurs blocs, soit une fonction.
- On alignera les déclarations avec des tab (sous Emacs M-i)
- Il est interdit d'affecter et de déclarer une variable en même temps, excepté lorsque l'on manipule des variables statiques ou globales
- Abuser des static pour faire des globales est interdit

```
1   {
2     static int toto = 15;
3     return (&toto);
4   }
```

- Le symbole de pointeur (*) porte toujours sur la variable (ou fonction), et jamais sur le type

```
1   t_scripts    *load_script(char *file_name)
2   {
3     FILE       *fd; /* correct */
4     t_scripts  *script_head; /* correct */
5     t_scripts  *last_script; /* correct */
6     static int pipot = 1; /* Correct */
7     char       *tmp = NULL; /* incorrect */
8     int*       cp; /* incorrect */
9
10    current_line = 0;
11    nbr_line = 0;
12    script_head = SCRIPT_END;
13    F_FOPEN(fd, file_name, "r", "script file");
14    ...
15  }
```

# Structures de contrôle

- Une structure de contrôle sera toujours suivi d'un retour à la ligne.

```
1   if (cp) return (cp); /* Incorrect */
2
3   if (cp) {return (cp);} /* Incorrect */
4
5   if (cp) /* Correct */
6     return (cp);
7
8   if (cp) /* Correct */
9     {
10      return (cp);
11    }
12
13  if (cp) /* Admis */
14  {
15      return (cp);
16  }
```

Selon sa configuration, Emacs indentera comme dans le quatrième ou cinquième exemple. Les deux sont acceptés. Pour les mêmes raisons, la taille du saut de l'indentation peut varier. On préférera 2 espaces.

# Paramètres

- La déclaration des paramètres se fera à la syntaxe ISO/ANSI C.
- Les virgules ne sont autorisées que dans ce contexte.
- Au maximum on doit trouver 4 paramètres dans une fonction. Pour passer plus d'information, faites une structure et passez un pointeur sur cette structure (et jamais la structure directement).

```
1   type func(type1 p1, type2 p2, type3 p3)
2   {
3   }
4
5   type func(type1 p1,
6             type2 p2,
7             type3 p3)
8   {
9   }
```

# Espaces

- Un espace derrière la virgule.
- Pas d'espace entre le nom d'une fonction et la '('.
- Un espace entre un mot clé C (avec ou sans argument) et la '('.
- Pas d'espace après un opérateur unaire.
- Tous les opérateurs binaires et ternaires sont séparés des arguments par un espace de part et d'autre.
- Il faut indenter les caractères qui suivent un #if ou un #ifdef.

'return' est un mot clé

'sizeof' n'est pas considéré comme étant un mot clé

```
1  if (!cp)
2    exit (1);
3  return (cp ? cp + 1 : sizeof(int));
4
5  #ifndef DEV_BSIZE
6  # ifdef BSIZE
7  #  define DEV_BSIZE BSIZE
8  # else /* !BSIZE */
9  #  define DEV_BSIZE 4096
10 # endif /* !BSIZE */
11 #endif /* !DEV_BSIZE */
```

# Mots clés interdits

- switch
- for
- break
- continue
- goto
- do while

Le mot clé `switch` est interdit car dans la plupart des cas il peut être remplacé par un tableau de pointeurs de fonctions.
Le mot clé `for` est interdit pour ne pas décentraliser l'initialisation des variables.
Les mots clé `break` et `continue` sont interdits pour ne pas casser l'exécution logique du programme.

# Headers

- La norme s'applique aussi aux headers.
- On les protégera contre la double inclusion. Si le fichier est foo.h, la macro témoin est FOO_H_
- Une inclusion de header (.h) dont on ne se sert pas est interdite.

# Makefile

- Les règles `$(NAME)`, `clean`, `fclean` et `all` sont obligatoires.
- Dans le cas d'un projet multi-binaires, en plus des règles précédentes, vous devez avoir une règle `all` compilant les deux binaires ainsi qu'une règle spécifique à chaque binaire compilé.
- Un exemple de makefile :

```
1
2   ##
3   ## Makefile
4   ##
5   ## Made by Astek
6   ## Login <astek@epitech.eu>
7   ##
8   ## Started on Fri Jan 28 11:52:30 2011 Astek
9   ## Last update Mon Apr 4 18:08:43 2011 Astek
10  ##
11
12  CC      = gcc
13
14  RM      = rm -f
15
16  CFLAGS  += -Wextra -Wall -Werror
17  CFLAGS  += -ansi -pedantic
18  CFLAGS  += -I.
19
20  LDFLAGS =
21
22  NAME    = putchar
23
24  SRCS    = main.c \
25            my_putchar.c
26
27  OBJS    = $(SRCS:.c=.o)
28
29
30  all: $(NAME)
31
32  $(NAME): $(OBJS)
33      $(CC) $(OBJS) -o $(NAME) $(LDFLAGS)
34
35  clean:
36      $(RM) $(OBJS)
37
38  fclean: clean
39      $(RM) $(NAME)
40
41  re: fclean all
42
43  .PHONY: all clean fclean re
```