Norme C-Epitech

Responsable Astek
astek_resp@epitech.eu

EPITECH.
L'ECOLE DE L'INNOVATION ET DE
L'EXPERTISE INFORMATIQUE

Table des matières

Qu'est-ce que c'est ?

- La norme C-Epitech est une convention de programmation qui a été créée 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, y compris pour les commentaires, 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 inclusions de headers (système ou non), les déclarations, les defines, les prototypes et les macros sont autorisés dans les fichiers headers.
- Toute inclusion de header doit etre justifiée.
- Les macros multilignes sont interdites.
- Les instructions multilignes sont tolérées.
- 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 prototypes de fonctions et 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).

Les 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 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.
- L'alignement du nom des variables avec le nom de la fonction est locale à la fonction, pas au fichier.

```
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  }
14
15  int                 address_exists(void *addr)
16  {
17      t_very_long_type **ptr;
18      int              i;
19
20      ptr = my_garbage_collector;
21      i = 0;
22      while (ptr[i])
23        {
24          if (ptr[i]->addr == addr)
25              return (TRUE);
26          i++;
27        }
28      return (FALSE);
29  }
```

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, de fonctions, de fichiers et de répertoires doivent être composés exclusivement de minuscules, de chiffres et de '_'.

```
1   void hello_world()
2   {
3   }
```

- Seuls les noms de macros sont en majuscules.

```
1   #define FOO "bar"
```

- Un typedef sur s_my_struct doit s'appeler t_my_struct.

```
1   typedef 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 d'union doit commencer par u_.
- Un nom de globale doit commencer par g_.
- Toute utilisation de variable globale doit etre jusitifiée.

Déclarations / Affectations

- 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). Cela est valable aussi pour les prototypes de fonction.
- 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. Toute variable statique doit etre justifiée.

```
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 suivie d'un retour à la ligne.

```
1   if (cp) return (cp); /* Incorrect */
2
3   if (cp) {return (cp);} /* Incorrect */
4
5   if (cp) { /* Incorrect */
6     return (cp);
7   }
8
9   if (cp) /* Admis */
10    return (cp);
11
12  if (cp) /* Correct */
13    {
14      return (cp);
15    }
16
17  if (cp) /* Admis */
18  {
19     return (cp);
20  }
```

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).

Ce n'est pas une raison pour mettre tout votre programme dans une seule structure ! N'oubliez pas qu'une structure doit contenir des informations qui sont liées à la même entité logique.

```
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 '(' ou le ';' dans le cas d'un return.
- 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.
- Il faut indenter les macros imbriquées.
- Une ligne vide doit séparer les définitions de fonctions.

'return' est un mot clé

'sizeof' n'est pas considéré comme étant un mot clé. Nous considérons 'sizeof' comme un cas particulier, son utilisation est similaire à un appel de fonction.

```
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
- 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.
Le mot clé `goto` est interdit pour ne pas casser l'exécution logique du programme.
Le statement `do while` est interdit parce que les étudiants l'utilise à mauvais escient.

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_

```
1 #ifndef FOO_H_
2 # define FOO_H_
3
4 #endif /* !FOO_H_ */
```

- Une inclusion de header (.h) dont on ne se sert pas est interdite.

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 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.
- Dans le cas d'un projet faisant appel à une bibliothèque de fonctions (par exemple une libmy), votre Makefile doit compiler automatiquement cette bibliothèque.
- L'utilisation de 'wildcard' (*.c par exemple) est interdite.
- 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 Wed Nov 27 17:09:04 2013 felix pescarmona
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
```