Norme d'écriture en C

Version du 11 December 2002

Auteurs Divers

Ce document est un aperçu, en français, du document censé uniformiser l'écriture des programmes C au sein de l'EPITA : le *Programming style standard*.

Il a pour double objectif d'aborder les diverses précisions du standard sous la forme d'un récapitulatif, et d'apporter une première approche en français pour les étudiants qui ne sont pas encore assez à l'aise avec l'anglais.

Il n'a pas pour objectif d'être un substitut au standard, et en cas de litige le standard prime.

Voici les sujets abordés :

- les conventions de nommage ;
- la présentation, au niveau global (fichiers) et local (blocs, lignes);
- les en-têtes et diverses informations annexes.

Pour éviter les problèmes d'interprétation, ce "résumé" est plus restrictif que le standard. En considérant interdites les constructions non détaillées ici, on est sûr de ne pas violer le standard.

Enfin, il est à noter que le standard contient des justifications sur certaines conventions qui pourront apparaître ici comme arbitraires.

Table of Contents

1 Conventions de nommage

1.1 Général

Les identifiants doivent être explicites ou mnémoniques.

Les abréviations sont tolérées dans la mesure o elles permettent de réduire la taille du nom sans en perdre le sens. Les parties des noms composites devraient être séparées par '_'.

Par exemple :

```
#define MAX_LINE_SIZE               1024
#define COMMENT_START_DELIMITER     '#'
#define MAX_FILE_NAME_LENGTH        2048
```

Tous les identifiants doivent être exprimés en anglais, sans faute d'orthographe.

1.2 Casse

Les noms de variables, de fonctions et de fichiers doivent être écrits en minuscule, et ne peuvent contenir que des lettres, des chiffres et le caractère '_'.

Les noms de macros sont écrits entièrement en majuscules.

Les arguments de macros prennent une majuscule en début, comme ceci :

```
#define XFREE(Var)      \
  do                    \
  {                     \
    if (Var)            \
      free(Var);        \
  }                     \
  while (0)
```

1.3 Types

Le nom d'une structure doit commencer par 's_', celui d'un "typedef" par 't_', 'u_' pour une union et 'e_' pour une énumération :

```
typedef unsigned char           t_cmap[COLOR_WIDTH * NCOLORS];
typedef unsigned char           t_pixel;

struct                          s_picture
{
  int                           width;
  int                           height;
  t_cmap                        cmap;
  t_pixel                       *picture;
};

typedef struct s_picture        t_picture;
```

Les noms de variables globales doivent commencer par 'gl_'.

2 Présentation locale

2.1 Indentation et espacements locaux

L'indentation autorisée est celle obtenue avec Emacs avec sa configuration standard.

```
int             get_type(char type_char)
{
  t_types       *types;

  for (types = types_tab; types->type_val; ++types)
    if (types->type_name == type_char)
      return types->type_val;
  return -1;
}
```

On passe toujours à la ligne après les accolades ou une structure de contrôle. On indente toujours après les accolades.

```
if (cp) return (cp);    /* Incorrect */

if (cp) {return (cp);}  /* Incorrect */

if (cp)                 /* Correct */
  {
    return (cp);
  }

if (cp)                 /* Correct */
{
  return (cp);
}
```

La quantité d'espaces d'indentation peut varier, mais on préférera deux espaces.

Il faut un espace après ‘`,`’ et ‘`;`’.

Il ne faut pas d'espace, ni avant, ni après, les opérateurs unaires préfixes et suffixes (‘`++`’, ‘`*`’, ‘`++`’ et ‘`--`’, ‘`-`’...).

Il faut un espace avant et après les opérateurs binaires, *y compris* les opérateurs d'affectation.

```
x+=10*++x; /* Incorrect */

x += 10 * ++x; /* Correct */
```

Les mots-clef du C doivent *toujours* être suivis par un espace, même avant un point-virgule. En revanche, lors d'un appel de fonction le nom de fonction est collé à la parenthèse ouvrante :

```
void    *xmalloc(size_t n)
{
  void  *p;

  p = malloc(n);
  if (p == 0)
  {
    fprintf(stderr, "Virtual memory exhausted.\n");
    exit(1);
  }
  return p;
}
```

Attention, '`return`' est un mot-clef, tandis que '`exit`' n'en est pas un.

2.2 Espacement des blocs

Une ligne ne doit pas dépasser 80 caractères de largeur.

Dans un bloc de fonction, les déclarations doivent être séparés du code par une ligne vide. Pour rajouter des espacements supplémentaires, il faut soit créer de nouveaux blocs, soit découper la fonctions en plusieurs fonctions, soit créer des macros.

```
int     foo(int fd, char **checkboard)
{
  int   i, j;
  char  c;

  for (i = 0; i < 8; ++i)
    for (j = 0; j < 8; ++j)
    {
      read(fd, &c, 1);
      checkboard[i][j] = c;
    }
}
```

On fera attention à avoir des fonctions courtes et claires, de 25 lignes maximum (entre les 2 accolades). On évitera aussi d'avoir plus de 5 fonctions par fichier, une serait l'idéal. Plus de précisions sur ces points se trouvent dans la section *Code density and documentation* du standard.

Les appels à la fonction '`assert`' ne sont pas comptabilisés dans le compte des lignes.

Il ne doit pas y avoir de commentaires dans les fonctions. Si il y a besoin de commentaires, ils doivent se trouver avant la fonction.

```
/*
** this function computes ....
*/
void    *dummy()
{
}
```

2.3 Déclarations et fonctions

Les fonctions devraient prendre moins de 4 arguments, qui doivent être déclarés avec la syntaxe ANSI :

```
type    func(type1 p1, type2 p2, type3 p3)
{
}

/* voire... */

type    func(type1 p1,
             type2 p2,
             type3 p3)
{
}
```

Pour passer plus d'informations à une fonction, il vaut mieux utiliser une structure et passer un pointeur vers une instance de cette structure.

En C, ce sont les variables qui pointent vers les données, il n'y a pas de "type pointeur". De ce fait, on accole '`*`' au nom de variable, et pas au type :

```
const char  *str; /* Correct */
const char*  str; /* Incorrect */
```

Il faut aligner les déclarations entre elles, et avec le nom de la fonction. On évitera aussi de déclarer plusieurs variables sur la même ligne.

Les variables ne peuvent pas être déclarées et initialisées en même temps, sauf s'il s'agit d'une variable locale avec l'attribut '`static`'.

```
int        foo()
{
  int      i;    /* Correct */
  int x; char y; /* Incorrect */

  i = 0; /* l'initialisation après la déclaration */
}
```

Les noms de champs dans une structure ou une union doivent être alignés avec le nom de la structure :

```
struct  s_point
{
  int   x;
  int   y;
  long  color;
};
```

De même, les valeurs d'énumération doivent apparaître chacune sur une ligne, alignées avec le nom de l'énumération :

```
enum    e_boolean
{
        b_true,
        b_false
};
```

Par ailleurs, les fonctions ne doivent pas renvoyer de structures ni d'unions, ni en prendre en argument.

2.4 Sauts et 'switch'

L'instruction '`goto`' est interdite.

On ne peut utiliser '`switch`' que sur des types énumération. et un bloc '`switch`' doit obligatoirement contenir le cas '`default`' s'il n'y a pas de '`case`' pour toutes les valeurs de l'énumération.

Par ailleurs, tous les blocs '`case`' du '`switch`' doivent se terminer par '`break`'.

Enfin, on indentera comme suit :

```
switch (c)
{
  case c_x:
    foo();
    break ;
  case c_y:
    bar();
    break ;
  default:
    break ;
}
```

On n'utilisera pas '`return`' dans un '`switch`', sauf sous certaines conditions très précises spécifiées dans le standard.

2.5 Directives préprocesseur et commentaires

Le symbole '`#`' doit apparaître sur la première colonne, et on préfèrera indenter les '`#if`' et '`#ifdef`' comme suit :

```
#ifndef DEV_BSIZE
# ifdef BSIZE
#  define DEV_BSIZE BSIZE
# else /* !BSIZE */
#  define DEV_BSIZE 4096
# endif /* !BSIZE */
#endif /* !DEV_BSIZE */
```

On préfèrera écrire les commentaires en anglais, sans faute. Un commentaire ne devrait pas comporter qu'une seule ligne (si le commentaire est court, c'est qu'il ne devrait même pas y avoir d'explication).

Les commentaires doivent être présentés comme suit :

```
/*
 * Incorrect
 */

/* Incorrect
 */

/*
** Correct
*/
```

3 Présentation globale

3.1 Fichiers en-tête et inclusions

Les indications de présentation locale sont applicables pour les fichiers en-tête (‘`.h`’).

Les fichiers en-tête doivent être protégés contre l’inclusion multiple, en utilisant une macro de même nom que le fichier, en majuscules. Par exemple, le fichier ‘`foo.h`’ sera protégé comme suit :

```
#ifndef FOO_H
# define FOO_H
/*
** Contents of foo.h
*/
#endif /* !FOO_H */
```

Les inclusions (directives ‘`#include`’) doivent apparaître au début d’un fichier. Les inclusions d’en-têtes système doivent apparaître *avant* les inclusions “locales” :

Incorrect :

```
#ifndef FOO_H
# define FOO_H

int bar();

# include "bar.h"

int foo();

# include <stdio>
#endif /* !FOO_H */
```

Correct :

```
#ifndef FOO_H
# define FOO_H

# include <stdio>
# include "bar.h"

int bar();
int foo();

#endif /* !FOO_H */
```

3.2 Prototypes

Toutes les fonctions exportées (à portée globale) doivent être correctement prototypées. Les prototypes doivent apparaître dans des fichiers en-tête, pas dans les fichiers source.

Quand un fichier définit une fonction exportée, il doit inclure l’en-tête qui contient le prtototype de la fonction.

Par exemple :

Fichier `my_string.h` :

```
#ifndef MY_STRING_H
# define MY_STRING_H

# include <stddef.h>

size_t my_strlen(const char *);
char   *my_strdup(const char *);

#endif /* !MY_STRING_H */
```

Fichier `my_strlen.c` :

```
#include "my_string.h"

size_t my_strlen(const char *s)
{
/* definition of my_strlen */
}
```

Fichier `my_strdup.c` :

```
#include "my_string.h"

char *my_strdup(const char *s)
{
/* definition of my_strdup */
}
```

Les prototypes doivent être écrits dans le style ANSI : ils doivent spécifier à la fois le type de retour et les types des arguments.

Prototypes incorrects :

```
foo();

bar(int, long);

int baz();
```

Prototypes corrects :

```
int  foo(int x);

void bar(int, long);

int  baz(void);
```

Lorsque plusieurs prototypes se suivent, les noms de fonctions doivent être alignés :

Incorrect :

```
unsigned int  strlen(const char *);
char *strdup(const char *);
```

Correct :

```
unsigned int  strlen(const char *);
char          *strdup(const char *);
```

Il ne doit pas y avoir plus d'une fonction exportée par fichier source. Il ne doit pas y avoir plus de 5 fonctions locales ("static").

4 Indications supplémentaires

4.1 En-têtes de documentation

Tous les fichiers source et fichier en-tête doivent débuter avec un en-tête de documentation, de la forme suivante :

```
/*
** <filename> for <project> in <part>
**
** Made by <author>
** Login   <login name@site>
**
** Started on  <date> <author>
** Last update <date> <author>
*/
```

4.2 Valeurs de retour

Les fonctions système et de la bibliothèque standard renvoient souvent des valeurs qui indiquent si l'opération a réussi ou non. Les cas d'erreur doivent être gérés, mais cette gestion ne doit pas "polluer" le code. On utilisera des fonctions personnalisées pour réaliser l'appel, comme celle-ci :

```
void    *xmalloc(size_t n)
{
  void  *p;

  p = malloc(n);
  if (p == 0)
  {
    fprintf(stderr, "Virtual memory exhausted.\n");
    exit(1);
  }
  return p;
}
```

4.3 Désactivation du code

Pour désactiver un bloc de code, il ne faut pas utiliser les commentaires. Il vaut mieux utiliser '`#if 0`' et '`#endif`'. Attention, un projet rendu ne doit pas contenir de blocs de code désactivés (il faut les faire disparaître avant le rendu).