Université Jean Monnet of Saint-Etienne

L2 Informatique

Programmation Impérative

Quelques propositions de règles de codage en language C

Préambule

Ce document présente quelques règles de codage en langage C ayant pour objectif de présenter du code clair et permettant de faciliter la compréhension, la relecture, le contrôle visuel, la maintenance ou encore le travail en équipe.

Les éléments présentés dans ce document n'ont pas de caractère obligatoire strict, mais ils donnent une idée des règles de codage qu'un programmeur doit suivre.

1 Jeu de caractères

Le jeu de caractères normalisé par le langage C est le suivant :

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
! " # % & ' ( ) * + , - . / :
; < = > ? [ \ ] ^ _ { | } ~
```

plus l'espace, et les caractères de commande représentant :

- le saut de ligne
- le retour chariot
- la tabulation horizontale
- la tabulation verticale
- le saut de page.

L'utilisation d'autres caractères (comme les caractères accentués, par exemple) invoque un comportement défini par l'implémentation.

Dans la pratique, la plupart des compilateurs acceptent les extensions courantes comme IBM PC8 et ISO 8859-1 (aussi appelés respectivement OEM et ANSI dans le monde MS-DOS/Windows). Mais ils peuvent cependant être sujets à des problème de portabilité.

2 Indentation

Il est capital de produire un code bien indenté. Il existe différente manière d'effectuer l'indentation, l'essentiel est de bien la respecter. L'idée est d'utiliser un éditeur permettant d'assurer la bonne indentation du code. Il est généralement préférable d'utiliser des espaces plutôt que des tabulations, l'effet d'une tabulation n'étant pas uniforme.

3 Commentaires

La version normalisée standard du langage C ne supporte que les commentaires de type `/* commentaire */`, cependant les versions plus récentes supportent des commentaires en fin lignes du type
`// commentaire fin de ligne`
La version standard est préférable sur l'on utilise des commentaires sur plusieurs lignes.

S'il est important de bien commenter, on pourra également se passer de commentaire en donnant un nom judicieux aux différents identificateurs, de manière à faciliter la lecture du code.

Si on veut suspendre provisoirement une portion de code, il souvent plus judicieux de ne pas utiliser de commentaires (il pourrait y avoir des commentaires imbriqués) mais d'utiliser des directives du préprocesseur (`#ifdef ... #endif` ou `#if ... #endif`)). Exemple :

```
#if 0
   /* Compteur */
   int cpt ;
#endif
```

4 Conventions de nommage

Une des façons d'obtenir du code clair est de s'en tenir à une convention de nommage des identificateurs qui soit cohérente et parlante. Vous trouverez ci-dessous quelques recommandations générales.

4.1 Utilisation du caractère souligné (underscore)

Le caractère souligné '_' peut être utilisé comme séparateur visuel dans les identificateurs. Bien que ce soit techniquement possible, il est conseillé de ne pas l'utiliser au début d'un identificateur, pour deux raisons principales :

- La plupart de ces identificateurs sont réservés à l'implémentation du langage
- L'aspect très inesthétique de ces identificateurs

4.2 Macros

Les macros ayant des usages multiples, il est difficile de faire une généralité. Dans les cas où elles représentent une constante, il est recommandé d'utiliser les majuscules. Dans les autres cas, le choix sera le même que pour l'identificateur qu'elle substitue. Cependant, si on cherche à insister sur le fait qu'une soi-disant fonction est en fait une macro, on peut utiliser les majuscules. En fait tout dépend du contexte.

4.3 Constantes

Il est couramment admis que les constantes *'vraies'* (macros, énumérations) doivent être écrites en majuscules. Les variables qualifiées `const` seront écrites comme des variables.

L'identificateur d'une constant dépend du contexte. Il sert souvent à définir les valeurs des propriétés d'un objet (valeurs particulières, états etc.) Il est recommandé d'utiliser un préfixe qui relie les constantes qui qualifient une même propriété.

```
/*
 * Valeurs possibles de l'etat du voyant
 *
 * OFF   : eteint
 * ON    : allume fixe
 * BLINK : clignotant
 */
enum
{
   VOYANT_STS_OFF,
   VOYANT_STS_ON,
   VOYANT_STS_BLINK,

   VOYANT_STS_NB
};
```

Il y a deux façons de créer des valeurs constantes. Avec une macro ou avec une énumération comme ci-dessus.

Pour les constantes numériques de type entier, il est recommandé d'utiliser les `enum`, qui présentent des avantages comme le typage, la numérotation automatique, ou l'interprétation textuelle automatique dans les débogueurs.

Pour les constantes numériques des autres types (réels, nombres non signés) et les chaînes, il n'y a pas le choix, il faut utiliser les macros. Ne pas oublier les qualificateurs de macros pour les valeurs numériques autres que entières : suffixes de macros numériques `u` ou `U` `unsigned` :

- `l` ou `L` : `long`
- `ul` ou `UL` : `unsigned long`
- `f` ou `F` : `float`

Ne pas oublier non plus que pour qu'une constante soit de type double, au moins un des membres doit être de type double, donc comprendre un '.' dans son expression :

```
#define PI (22/7)
```

est un entier qui vaut 3.

```
#define PI (22/7.0)
```

est un double qui vaut environ 3.14

4.4 Types

Le langage C permet de créer un alias sur un type existant plus ou moins complexe ou un autre alias existant à l'aide du mot clé `typedef`.

Il est d'usage d'utiliser les minuscules. Il est recommandé d'utiliser des suffixes tels que :

- `_e` : `typedef enum`
- `_u` : `typedef` union
- `_s` : `typedef` struct
- `_a` : tableau (array)
- `_f` : fonction

Note : il est recommandé de ne pas utiliser le suffixe '`_t`', car il est réservé par la norme `POSIX` pour définir des alias de types.

4.5 Objets

Il est d'usage d'utiliser les minuscules.

On recommande parfois que la longueur des identificateurs des objets et des fonctions soit proportionnelle à leur portée.

Les variables introduisent en plus la notion de durée de vie (duration). Il est bon qu'au premier coup d'oeil, on sache exactement à quoi on a affaire. Il est bon de pouvoir aussi identifier rapidement les variables en fonction de leur propriétés. Les plus remarquables sont :

- la variable simple
- le pointeur
- le tableau statique
- le tableau dynamique
- la chaîne de caractères.

Il est alors proposé d'utiliser les préfixes suivants :

(vide)	variable simple
`p_`	pointeur
`a_` ou `sa_`	tableau statique (static array)
`da_`	tableau dynamique (dynamic array)
`s_`	chaîne de caracteres (string)

En ce qui concerne la durée de vie et la portée voici des préfixes associés :

	Portée	Durée de vie
(vide)	bloc	bloc
`g_`	bloc	programme
`S_`	module	programme
`G_`	programme	programme

4.6 Fonctions

Il est d'usage d'utiliser les minuscules. On a parfois besoin de 2 mots pour nommer une fonction. Il n'est évidemment pas question d'accoler ces deux mots en minuscule, car le code serait vite illisible. Il existe 2 pratiques répandues :

- Coller les mots en mettant une majuscule au début de chaque mot :

```
OuvrirFichier()
```

  ou (variante 'à-la-Java') à partir du 2ème mot :

```
ouvrirFichier()
```

- Séparer les mots en mettant un souligné (underscore) entre chaque mot.

```
ouvrir_fichier()
```

C'est une question de goût, le principal est d'être clair et cohérent.

Pour une fonction, on choisit plutôt un identificateur qui exprime une action. (Verbe, verbe + substantif)

5 Organisation du code source

Il est pratique d'adopter une disposition logique et cohérente dans les fichiers sources et d'en-têtes. Il est d'usage d'utiliser un principe simple, qui consiste à définir ce que l'on doit utiliser. C'est pourquoi la disposition suivante est recommandée.

5.1 Fichiers sources (*.c)

Liste ordonnée des éléments pouvant être contenus dans un fichier source

- Inclusion des en-têtes (.h) nécessaires
- Définition des macros privées
- Définition des constantes privées
- Définition des types privés
- Définition des structures privées
- Définition des variables globales privées
- Définitions des fonctions privées
- Définition des fonctions publiques
- Définition des variables globales publiques

(le mot clé privé fait référence à l'utilisation du mot clé `static`.[1]

5.2 Fichiers d'en-tête (*.h)

Règles d'or régissant la définition des fichiers d'en-tête

- Un fichier d'en-tête doit être protégé contre les inclusions multiples dans la même unité de compilation.
- Un fichier d'en-tête doit être autonome.
- Un fichier d'en-tête ne doit inclure que le strict nécessaire à son autonomie.

Liste ordonnée des éléments pouvant être contenus dans un fichier d'en-tête

- Inclusion des headers nécessaires et suffisants
- Définition des macros publiques
- Définition des constantes publiques
- Définition des types publiques
- Définition des structures publiques
- Déclaration des fonctions publiques
- Déclaration des variables globales publiques.

[1] Sur une variable locale, il permet de définir une variable maintenue pendant toute la durée d'exécution du programme (comme les variables globales) seulement, puisqu'elle a été déclarée à l'intérieur d'une fonction, sa visibilité sera limitée au corps de cette fonction. Sur une variable globale, le modificateur `static` permet de réduire la visibilité de la variable au fichier où elle est déclarée.

Protection contre les inclusions multiples Les fichiers d'en-tête pouvant être inclus dans des fichiers sources, comme dans des fichiers d'en-tête, et ce, dans un ordre non spécifié, il est indispensable de se prémunir contre les risques d'inclusions multiples dans la même unité de compilation.

```
#ifndef IDENTIFICATEUR
#define IDENTIFICATEUR

/* zone protegee contre les inclusions multiples */

#endif
```

Le principe de protection étant basé sur la définition d'une macro, cette macro doit être unique afin d'éviter les protections abusives. Des exemples seront donnés dans les exemples du cours.

5.3 Création d'archive

Afin de bien organiser les fichiers, je conseille de créer des répertoires pour les sources et pour les fichiers de sortie:

Type	Répertoire	Fichiers
Génération	`./`	Makefile
Source	`./inc/`	en-tête `*.h`
Source	`./src/`	source `*.c`
Sortie	`./obj/`	objets intermédiaires `*.o`
Sortie	`./bin/`	exécutable

Consulter les exemples de Makefile sur la page du cours.