EISTI	**Cycle préparatoire 1ère année (Campus de Pau)** **Normes de programmation** *Florent* Devin
Matière : **Informatique**	*Date* : **Octobre 2019**
	Durée :
	Nombre de pages : **6**

1 Introduction

Ce document fixe un certain nombre de règles élémentaires (appelées normes de codage) dans le but d'aider le développeur à écrire des programmes C[1] de manière propre et compréhensible. L'informatique s'industrialise, et tend à normaliser les processus de codage, de test, de conduite de projet, ... Les normes de codage ont pour objectifs d'améliorer la lisibilité du code, de permettre une lecture rapide du code, de faciliter la maintenance, ... Dès que l'on développe un projet à plusieurs, il est important de se mettre d'accord sur des règles de codage. Les normes de C ne précisent pas les conventions de codage. Il est donc important de fixer un cadre afin que vous puissiez travailler ensemble plus facilement.

2 Organisation des programmes

Les programmes C doivent être écrits selon le principe de la programmation modulaire. Est appelé un module l'ensemble `<fichier interface - fichier implémentation>` regroupant les fonctions traitant d'un seul et même sujet (manipulation d'une structure, thème, ...). Par exemple, le module *Pile* est constitué d'un fichier interface `Pile.h` et d'un fichier `Pile.c`. Un fichier *indépendant* contiendra la fonction `main()` qui lancera l'application. La fonction `main` ne peut avoir que trois (3) formes :

Code
```
int main (int argc, char* argv[])
```

Code
```
int main (int argc, char** argv)
```

Code
```
int main (void)
```

Le fichier interface, lorsqu'il existe, contiendra les définitions des structures de données, les définitions des constantes symboliques (`#define`), les prototypes des fonctions exportées, ...

1. Ces règles sont majoritairement applicable pour d'autres langages

3 Compilation

L'utilisation d'un `Makefile`[2] est obligatoire, lorsque vous compilez plus d'un fichier. Il s'appellera obligatoirement `Makefile`, et comportera au minimum les cibles `all`, `clean`, `doc`. Dans ce `Makefile`, vous veillerez à utiliser les options de compilation `-Wall -Wextra -pedantic`

4 Règles d'écriture

4.1 Limiter la taille

Afin de permettre une lecture rapide, une fonction doit être courte. Les normes de codage fixent donc une limite en nombre de lignes pour les fonctions, pour les modules. Si une fonction est trop longue, c'est que son traitement a été mal défini, ou trop large. Il faut donc la découper en plusieurs sous-fonctions plus simples. Le même principe s'applique aussi pour les modules.

À l'EISTI, nous avons choisi de limiter les fonctions à vingt-cinq (25) lignes, et les modules à cinq-cents (500) lignes. Une fonction commence toujours par une déclaration, ce qui fixe le début de la fonction ; et se termine toujours par une `}`. Le nombre de lignes est calculé par la formule : fin − début + 1. Les commentaires décrivant la fonction n'interviennent pas dans le comptage du nombre de lignes, en toute logique, car ils doivent être situés avant le début de la fonction.

La taille d'une ligne est fixée à cent-cinquante caractères.

4.2 Accolades et indentation

Les accolades en C jouent le rôle de séparateur de blocs. Il faut donc avoir des règles, qui régissent celle-ci, claires et précises. Chaque nouveau bloc, donc entre deux accolades `{...}` doit être indenté, en utilisant une tabulation[3]. Il n'existe pas d'autres raisons pour utiliser des tabulations[4]. Nous avons choisi les règles d'indentation proposée par Kernighan et Ritchie. Une accolade ouvrante (`{`) se trouve toujours à la fin d'une ligne, et est séparée d'un espace ; une accolade fermante (`}`) doit se trouver en premier sur une ligne (dans le même niveau de l'indentation de bloc).

On écrit donc

Code

```
if (x) {
        we do y
}
```

Et pas

Code

```
if (x)
{ we do y }
if (x) {we do y}
if (x)
        { we do y }
```

2. Le `Makefile` fera l'occasion d'un autre cours
3. En général, une tabulation fait 8 caractères
4. Ce qui signifie que vous devez toujours créer un bloc quand cela est nécessaire

Il y a cependant une exception : les fonctions. Leur accolade ouvrante se trouve sur le début de la ligne suivante :

Code

```
int function (int c)
{
        body of function
}
```

Les accolades fermantes sont sur une ligne seule, sauf si elles font partie intégrante d'une instruction, comme :

Code

```
if (x == y) {
        ...
} else {
        ...
}

do {
        ...
} while (condition);
```

5 Règles de nommage

- Les noms des identificateurs et des fichiers utilisent la même langue.
- Les noms de variables et de fonctions sont *significatifs* [5].
- Les noms `i`, `j`, `k` sont tolérés pour les boucles uniquement.
- Les noms variables, fonctions, ... sont en *camel case* [6], ou en *snake case* [7]. Une préférence sera accordée au *camel case*.
- Les constantes s'écrivent toujours en majuscule : `#define TAILLE_TABLEAU 10`
- Les noms de paramètres formels et noms de variables locales utilisent les mêmes règles qui ci-dessus.
- On pourra aussi utiliser les conventions suivantes :
  - Les types personnels sont postfixés par `_t` : `typedef int entier_t;`
  - Les pointeurs sont préfixées par `p_` : `int* p_nb;`
  - Les pointeurs vers des tableaux sont préfixés par `t_` ou `tab` : `int* tabTermes;`
  - Les variables sont préfixées par une lettre ou un groupe de lettre représentant leur type : `int i_nb; char* str_adresseMail`
- Les noms de module sont en *pascal case* [8].

6 Règles de codage

- Toutes les fonction sont explicitement typées.
- Les déclarations de type et de variables sont effectuées séparément.

5. qui indiquent clairement le concept nommé
6. Les mots sont en minuscule, liés sans espace. Tous les mots commencent par une majuscule, sauf le premier mot. Exemple : `maVariable`, `adresseMail`, ...
7. Tous les mots sont en minuscules, liés par des *underscores* (_). Exemple : `ma_variable`, `adresse_mail`
8. Les mots sont en *initiale majuscule* liés sans espaces. Exemple : `GestionDesUtilisateurs`

— Les dimensions d'un tableau sont des constantes symboliques.
— L'utilisation de variables globales est interdite à moins que le programmeur n'ait d'autre solution [9].
— L'instruction **`goto`** est interdite.
— L'affectation multiple est interdite.
— Chaque unité de traitement d'un **`switch`** est terminée par un **`break`**.
— Toute instruction **`switch`** contient une branche **`default`** même vide.
— L'instruction **`switch`** est préférée à l'instruction **`if ... else if ... else`** si les deux sont applicables.
— L'instruction **`for`** est préférée à l'instruction **`while`** pour les traitements itératifs basés sur un compteur.
— Pas d'affectation dans une condition.
— Les fonctions de type différent de **`void`** terminent leur exécution par un **`return`** `(val);`.
— Les expressions sont bien parenthésées.
— Tous les cas d'erreur sont testés et traités.
— Une fonction retournant un code d'erreur doit retourner une valeur différente pour chaque type d'erreur.
— La valeur d'un indice de boucle n'est pas réutilisée en dehors de la boucle.

7 Commentaires

— Commenter le code *pendant* l'écriture du code. Ne commentez pas après, cela sera trop tard.
— Un module commence toujours par un cartouche qui le décrit.
— Chaque fonction est précédée d'un cartouche décrivant son rôle (contrat), ses paramètres d'entrée et sa(ses) valeur(s) de retour.
— Aucun commentaire ne se trouve sur la même ligne qu'une instruction, excepté pour commenter une déclaration.
— Les commentaires décrivent les fonctions du code et non le code lui-même (ce que fait la fonction pas le code).
— Au maximum une variable est déclarée par ligne.
— Les commentaires sont tous dans la même langue.
— Au maximum un champ de structure est déclaré par ligne.
— Un commentaire suit chaque champ de structure.
— Un commentaire suit chaque déclaration de variable.
— Les commentaires doivent être rédigé au format *doxygen*

7.1 Cartouches

Un commentaire commente ce que fait le code, et non pas le code lui-même. Il n'est en aucun cas la traduction du code, mais la traduction de la pensée du développeur.

Tout fichier doit commencer par un cartouche.

Un cartouche, ou entête, est constitué :
— d'une explication sur le contenu du fichier ;
— du nom du ou des auteurs ;
— de la date de création du fichier ;
— de la date de dernière modification du fichier ;
— et d'un numéro de version.

Pour un fichier, on aura donc :

9. Une justification est alors nécessaire

Code

```
/*!
\file nomDuFichier.c
\brief Courte description
\author Florent Devin <fd@eisti.eu>
\author Un autre auteur ayant participé à l'écriture du
    code de ce fichier <toto@eisti.eu>
\version 0.1
\date 4 novembre 2019

Une description complète de ce que fait ce module

*/
```

Toute fonction doit être commentée pour pouvoir plus facilement être :
— déboggée (correction des erreurs).
— appelée par une autre fonction de bonne façon.

Dans cette optique, on doit insérer un commentaire juste avant le prototype de fonction. Ce commentaire doit comprendre :
— La signification de chaque paramètre.
— Des préconditions : ensemble des conditions (evt. vide) que doivent vérifier les paramètres pour que le fonction s'exécute dans le cadre prévu par le programmeur de cette fonction.
— Ce que fait la fonction.
— Ce que retourne la fonction si elle retourne un résultat.
— L'auteur de la fonction. En général, il n'y a qu'un seul auteur, car une fonction est rarement codé par deux personnes différentes.

Par exemple, la fonction `main` se document de la sorte :

Code

```
/**
* \fn int main(int argc, char** argv)
* \brief Fonction principale.
* \author Florent Devin <fd@eisti.eu>
* \version 0.1
* \date 28 octobre 2019
* \param argc Nombre d'argument de la ligne de commande
* \param argv Paramètres de la ligne de commande
* \return 0 dans tous les cas
*
* Cette fonction affiche "Hello world" et quitte le programme
*
*/
```

En suivant ces règles, vous pouvez générer une documentation technique avec le programme `doxygen`.

8 Règles de structuration

— Un module est toujours composé d'une partie interface (`.h`) et d'un corps (`.c`).
— Toutes les variables ou fonctions exportées par un module sont déclarées dans son interface.

— Le corps d'un module fait explicitement référence à son interface.
— L'interface d'un module est auto-protégée contre les inclusions multiples :

Code

```
#ifndef INTERFACE_PILE
#define INTERFACE_PILE
...
interface de l'objet
...
#endif
```

9 Déverminage

Le déverminage est l'affaire du programmeur mais en aucune manière celui de l'utilisateur. Vous ne devez donc en aucun cas laisser des messages de déverminage lors de la remise d'un programme.