Conventions de codage pour le langage C

Table des matières

Introduction

Pourquoi des normes de programmation ?

- 80% des coûts d'un logiciel au cours de son cycle de vie vont à sa maintenance.
- Il est très rare qu'un logiciel soit maintenu durant toute son existence par son auteur original.
- Les normes de programmation améliorent la lisibilité du code, ce qui permet à d'autres informaticiens et programmeurs de le comprendre mieux et plus rapidement.

De telles normes favorisent en outre :

- l'écriture de programmes corrects,
- l'obtention d'un style homogène dans le code,
- la détection rapide des erreurs les plus courantes.

Ce document présente les conventions de codage en langage C appliquées **au département informatique de l'IUT**. Vous serez peut-être amenés à utiliser d'autres règles en entreprise.

Pour qu'une norme fonctionne, chaque personne qui écrit du code doit s'y conformer. Tout le monde, enseignants comme étudiants.

1. Présentation d'un programme source

1.1 Structure générale

- Un fichier source peut contenir plusieurs sections logiques (ex : la section de déclaration des constantes, la section de déclaration des variables, etc.). Ces sections doivent être séparées par une ligne vide (*cf.* 5.1) et chaque section doit débuter par un commentaire (*cf.* 2).

Structure type d'un fichier source en C

- *entête du programme (cf. 1.4)*
- *déclaration des fichiers inclus (les "#include")*
- *déclaration des constantes*
- *déclaration des types*
- *déclaration des procédures et fonctions (les prototypes)*
- *définition de la fonction main*
- *définition des procédures et fonctions*

(voir exemple complet en annexe)

1.2 Indentation

L'indentation est **obligatoire**. Le nombre d'espaces importe peu, le seul critère est la lisibilité des parties logiques d'un code au simple coup d'oeil.

1.3 Longueur des lignes

Une ligne de code ne doit pas dépasser 80 caractères. Il ne faut pas être obligé de faire défiler horizontalement le programme source pour voir la fin d'une ligne de code.

On peut écrire une instruction sur plusieurs lignes en respectant les règles suivantes :

- changer de ligne après une virgule,
- changer de ligne avant un opérateur,
- aligner la nouvelle ligne avec la ligne précédente,
- indenter si nécessaire.

Autre possibilité : découper l'instruction longue en plusieurs instructions.

```
    nomLong2 = nomLong2 * (nomLong3 + nomLong4
    - nomLong5) - 4 * nomLong6; // A EVITER

    nomLong1 = nomLong2 * (nomLong3 + nomLong4 – nomLong5)
              - 4 * nomLong6; // PREFERABLE

    nomLong = (nomLong3 + nomLong4 – nomLong5);
    nomLong2 = nomLong2 * nomLong - 4 * nomLong6; // PREFERABLE
```

1.4 Entête d'un programme

Chaque fichier doit comporter un en-tête minimal qui respecte la présentation suivante :

```
/**
* @file <nom du fichier source>
* @brief <brève description du programme>
* @author <nom de l'auteur du programme>
* @version <numéro de version>
* @date <date de création du programme>
*
* <description plus complète du programme>
*
*/
```

Exemple :

```
/**
* @file exemple.c
* @brief Programme illustrant les conventions de codage
* @author Arthur
* @version 1.0
* @date 01/09/2026
*
* Exemple de programme qui illustre les conventions de codage en C
* utilisées au département informatique de l'IUT de Lannion
*
*/
```

1.5 Enoncés d'importation (`#include`)

Il ne faut jamais spécifier de chemin d'accès complet dans les directives `#include`.

```
#include "c:\dev\include\NomBibliotheque.h" // A EVITER

#include "NomBibliotheque.h" // OK
```

Utiliser la directive `#include <NomBibliotheque.h>` pour des bibliothèques fournies par le fabricant du compilateur.

2. Commentaires

2.1 Styles de commentaire

Les programmes C peuvent avoir deux styles de commentaires.

Commentaire sur une ligne...

```
     if (bof > 1){
          // cas particulier
          res = TRUE;
     } else {
          // ne fonctionne que pour a impair
          res = isPrime(a);
     }
     return res
```

… ou bien commentaire de fin de ligne

```
     if (bof > 1){
          res = FALSE; // Expliquer pourquoi ici.
     }
```

Commentaire en bloc :

```
/*
     Ceci est
     un bloc de
     commentaire
*/
```

Toutes les lignes du commentaire sont encadrées par /* et */

2.2 Commentaires des procédures et fonctions

Utiliser obligatoirement des commentaires au début de chaque procédure/fonction. Ces commentaires doivent apparaître juste avant la définition de la procédure/fonction et doivent informer sur son objectif, sur chacun des paramètres et sur le résultat (s'il y en a). Ces informations seront fournies de la manière normalisée suivante :

```
     /**
     * @brief <brève description de la procédure ou fonction>
     * @param <nom, type, Entrée ou Entrée/Sortie et rôle>
     * @param ...
     * @param ...
     * @param ...
     * @param ...
     * @return <résultat : type et valeurs prises selon les cas>
     */
```

Une ligne par paramètre

Exemple :

```
/**
*
* @brief  Fonction d'ajout d'un nouvel élément
* @param  nouveau de type element, Entrée : l'élément à ajouter.
* @return AJOUT_OK si pas d'erreur, AJOUT_ERROR si l'élément existe
  déjà.
*
*/
int ajouterElement(element nouveau){
...
}
```

2.3 Pertinence des commentaires

Les commentaires permettent de comprendre l'objet d'un programme sans avoir à se "plonger dans le code". Dans tous les cas, il faut s'assurer que les commentaires sont **exhaustifs**, **pertinents**, **concis** et **non redondants**.

Exemples de commentaires **non pertinents** :

```
int total; // variable entière représentant le total

i = i + 1; // on incrémente i de 1
```

3 . Déclarations

3.1 Nombre de déclarations de variable par ligne

Ne déclarer qu'une seule variable par ligne parce que cela encourage les commentaires.

Exemple :

```
int niveau = 1;          // profondeur de l'arbre
int taille = TAILLE_MAX; // taille du tableau
```

est préférable à :

```
int niveau, taille;
```

3.2 Initialisation

Initialiser les variables là où elles sont déclarées.

Ne pas utiliser l'affectation dans une expression.

Ne pas mettre des nombres dans le code (exceptés 0 et 1 éventuellement). Définir plutôt des constantes.

Par exemple, **ne pas faire** :

```
char buffer[256];
for (int i = 0 ; i < 256 ; i++){
     buffer[i] = '.';
}
```

mais plutôt :

```
#define LONGUEUR_MAX 256;
#define CHAR_POINT '.';

char buffer[LONGUEUR_MAX];
for (int i = 0 ; i < LONGUEUR_MAX ; i++){
     buffer[i] = CHAR_POINT;
}
```

Ou encore, **ne pas faire** :

```
prix = prixBrut * 1.156; // pourquoi 1.156 ?
```

mais plutôt :

```
#define MARGE_COMMERCIALE 1.156;
...
prix = prixBrut * MARGE_COMMERCIALE;
```

3.3 Position

Placer les déclarations seulement **au début** d'une fonction (et pas au milieu du code).

Exemple :

```
int main(){
     int valeur1
     int valeur2;
     char carLu;

     <instructions>
}

void calculer(){
     int i;
     int nbTotal;

     <instructions>
}
```

3.4 Structures

En ce qui concerne les structures, utiliser toujours l'instruction "`typedef`".

Le nom du nouveau type doit commencer par t_

```
typedef struct{
     int elements[TAILLE_MAX];    // tableau contenant les entiers
     int nbElements;              // nombre d'entiers effectivement présents
} t_ensemble;
```

4. Instructions et blocs

4.1 Instructions simples

Chaque ligne doit contenir au plus une instruction.

```
argv++;         // CORRECT
argc--;         // CORRECT
argv++; argc--; // A PROSCRIRE
```

4.2 Instruction `return`

Toujours faire en sorte que le mot `return` ne soit présent qu'à la fin de la fonction. Cela rend la fonction plus facile à maintenir et à comprendre.

Le résultat d'une fonction doit toujours être transmis par une variable.

4.3 Bloc d'instructions

Un bloc d'instructions est composé d'une liste d'instructions ou de blocs encapsulés entre accolades. Les instructions ou blocs encapsulés doivent être indentés d'un niveau de plus que le bloc d'instructions.

Les accolades sont utilisées autour de chaque bloc, **même s'il n'est composé que d'une seule instruction**. Ceci facilite l'ajout d'instructions sans créer accidentellement des bugs dus à l'absence d'accolades.

L'accolade ouvrante doit être à la fin de la première ligne du bloc ; l'accolade fermante doit commencer une ligne et être indentée au même niveau que le début du bloc correspondant.

4.4 Blocs `if` ou `if-else`

Le bloc `if` doit être présenté comme suit :

```
if (<condition>){
     <instructions>
}
```

Le bloc `if-else` doit être présenté comme suit :

```
if (<condition>){
     <instructions>
} else {
     <instructions>
}
```

ou encore :

```
if (<condition>){
     <instructions>
} else if (<condition>){
     <instructions>
} else {
     <instructions>
}
```

4.5 Bloc `for`

Un bloc `for` doit avoir l'allure suivante :

```
for (<initialisation> ;  <condition> ; <incrémentation>){
     <instructions>
}
```

4.6 Bloc `while`

Un bloc `while` doit avoir l'allure suivante :

```
while (<condition>){
     <instructions>
}
```

4.7 Bloc `switch`

Un bloc `switch` doit avoir l'allure suivante :

```
switch (<variable>){
     case 'A':
          <instructions>
          break;
     case 'D':
          <instructions>
          break;
     case 'X':
          <instructions>
          break;
     default:
          <instructions>
          // break sur la dernière ligne non obligatoire
}
```

Toujours mettre un "`break`" à la fin de chaque "`case`". En effet, on peut facilement oublier le "`break`" par inadvertance sans objection de la part du compilateur.

Toujours terminer par la clause "`default`".

Le bloc switch est le seul cas d'utilisation de l'instruction break !

5. Espaces vides

5.1 Lignes vides

Les lignes vides améliorent la lisibilité en séparant des sections de code qui sont logiquement reliées. Utiliser une ligne vide :

- entre les procédures ou fonctions,
- entre les variables locales d'une procédure ou fonction et sa première instruction,
- avant un commentaire en bloc ou un commentaire sur une ligne.

5.2 Espaces

- Un mot réservé et une parenthèse doivent être séparés par un espace.

Exemple :

```
    while (maVariable < 0)
```

- On doit toujours utiliser un espace après les virgules dans une liste d'arguments.

- Les opérateurs doivent être séparés de leurs opérandes par un espace.

Exemples :

```
    a = c + d;
    a = (a + b) / (c * d);
```

- Les expressions dans un bloc `for` doivent être séparées par des espaces.

Exemple :

```
    for (i = 0 ; i < TAILLE_MAX ; i++){
         ...
```

6. Nomenclature

Les conventions de nomenclature rendent les programmes plus compréhensibles en les rendant plus faciles à lire. Elles peuvent aussi fournir de l'information quant au rôle d'un identificateur[1] et indiquer par exemple s'il s'agit d'une constante, d'une variable, etc.

Dans tout identificateur, il faut éviter d'utiliser des caractères accentués.

6.1 Notations

Deux notations sont couramment admises pour les identificateurs :

- la notation **camelCase**[2] où l'identificateur est écrit en minuscules, sauf pour l'initiale des mots qui le composent à l'exception du premier.

  Exemples :

```
maLargeurDeCarreau

couleurDeCarte

testDeLongueur
```

- la notation **snakeCase**[3] où les mots qui constituent l'identificateur sont séparés par un _

  Exemples :

```
ma_largeur_de_carreau

couleur_de_carte

test_de_longueur
```

6.2 Variables

Les noms de variables doivent être significatifs. Le simple fait de lire le nom d'une variable doit renseigner sur son rôle. Alors il ne faut pas hésiter à choisir comme nom de variable un ensemble de mots.

Pour les variables, utiliser la notation **camelCase**.

Exemples :

```
int nb;              // A EVITER
int nombreElements; // PREFERABLE
```

1 Un identificateur est le nom que l'on donne à une variable, une constante, une procédure, une fonction, *etc.*

2 Le terme `camelCase` vient de l'apparence des mots écrits dans ce style, où les majuscules au milieu des mots ressemblent aux bosses d'un chameau.

3 Le terme `snake_case` vient de l'apparence visuelle des traits de soulignement qui ressemblent à un serpent.

Le nom d'une variable booléenne doit être formulé comme une proposition.

Exemples :

```
bool estVide;
bool isOK;
```

6.3 Constantes symboliques

- Les noms de constantes symboliques doivent être en **majuscules**.
- Pour les constantes symboliques, utiliser la notation **snakeCase**.

Exemples :

```
#define LARGEUR_MIN 4
#define TAILLE_MAX 100
#define VRAI 1
#define FAUX 0
```

6.4 Procédures et fonctions

- Le nom d'une procédure ou d'une fonction doit commencer par un **verbe** et doit donner une indication sur son rôle.
- Utiliser la notation **snakeCase.**

Exemples de prototypes

```
void executer();
void afficher_menu();
void regler_heure(int heure);
int obtenir_heure();
```

6.5 Nouveaux types

- Le nom d'un nouveau type sera précédé de **t_**
- Pour les nouveaux types, utiliser la notation **snakeCase**

Exemples

```
typedef int t_tableau_entiers[TAILLE];

typedef float t_tab_reels[MAX];
```

7. Parenthèsage

Il est recommandé d'utiliser les parenthèses dans des expressions impliquant des opérateurs mixtes pour éviter les problèmes de précédence des opérateurs. Le lecteur ne connaît pas forcément les règles de précédence aussi bien que le programmeur original.

Exemples :

```
    if (a == b && c == d)        // A EVITER

    if ( (a == b) && (c == d) ) // OK


    total = prixUnitaire * quantité - ristourne;       // A EVITER

    total = (prixUnitaire * quantité) - ristourne;     // OK
```

Annexe

```c
/**
* @file exemple.c
* @brief Programme illustrant les conventions de codage
* @author Robert
* @version 2.0
* @date 15/09/2025
*
* Extrait d'un programme qui illustre les conventions de codage en langage C
* utilisées au département Informatique de l'IUT de Lannion.
* Le programme consiste à gérer un ensemble d'entiers.
*
*/

/* Fichiers inclus */
#include <stdio.h>
#include <stdlib.h>

/* Taille maximale d'un ensemble */
#define TAILLE_MAX 100

/* Structure utilisée pour implémenter la notion d'ensemble */
typedef struct{
      int elements[TAILLE_MAX]; // tableau contenant les entiers
      int nbElements;           // nombre d'entiers effectivement présents
} t_ensemble;

/* Enumération des différentes erreurs possibles lors de l'ajout d'une valeur */
typedef enum{
      ENSEMBLE_PLEIN,     // la taille maximale est atteinte
      VALEUR_EXISTANTE,   // la valeur à ajouter existe déjà
      OK                  // pas d'erreur
} t_codeErreur;

/* Déclaration des fonctions */
void initialiser_ensemble(t_ensemble *);
t_codeErreur ajouter_element(t_ensemble *, int);

/**
* @brief Entrée du programme
* @return EXIT_SUCCESS : arrêt normal du programme
* Initialise un ensemble puis lui ajoute une valeur lue au clavier
* en testant différents cas possibles
*
*/
```

```c
int main(){
	t_ensemble monEnsemble; // l'ensemble de travail
	int maValeur; // la valeur à ajouter à l'ensemble
	t_codeErreur resultatErreur; // code d'erreur suite à la tentative d'ajout
	initialiserEnsemble(&monEnsemble);
	printf("\n Donnez une valeur entière : ");
	scanf("%d", &maValeur);
	resultatErreur = ajouterElement(&monEnsemble, maValeur);
	if (resultatErreur == ENSEMBLE_PLEIN){
		printf("\n L'ensemble est plein !\n");
	}else if (resultatErreur == VALEUR_EXISTANTE) {
		printf("\n La valeur existe déjà !\n");
	}
	return EXIT_SUCCESS;
}

/**
* @brief Procédure d'initialisation d'un ensemble
* @param ens de type ensemble E/S : l'ensemble à initialiser
*
*/
void initialiser-ensemble(t_ensemble * ens){
	ens->nbElements = 0;
}

/**
* @brief Fonction d'ajout d'une valeur à un ensemble
* @param ens de type ensemble E/S : ensemble auquel ajouter la valeur
* @param valeur de type entier E : la valeur à ajouter
* @return ENSEMBLE_PLEIN si limite atteinte, VALEUR_EXISTANTE si valeur existe, OK
* sinon
*
*/
t_codeErreur ajouter_element(t_ensemble * ens, int valeur){
	t_codeErreur erreur = OK;
	if (ens->nbElements >= TAILLE_MAX) {
		erreur = ENSEMBLE_PLEIN;
	}else if (existe(*ens, valeur)){    // fonction 'existe' non fournie ici
		erreur = VALEUR_EXISTANTE;
	} else{
		ens->elements[ens->nbElements] = valeur;
		ens->nbElements++;
	}
	return erreur;
}
```