Conventions de codage

Langage C

Nom	C280116
Version	0.9.1
Rédacteurs	Frédéric Sananes Benoit Briatte

Sommaire

Objectif du présent document
Organisation des projets
Conventions de nommage
Modèles d'organisation de fichier
Indentation et formatage
Déclarations
Formatage des instructions structurées
Fonctions
Structures et unions
Commentaires

Objectif du présent document

Les conventions de codage forment un ensemble de règles adoptées par un groupe, les membres d'un projet, une entreprise de développement ou encore une institution comme une école, pour construire un code lisible, maintenable et compréhensible rapidement. Il est donc question dans ce document de définir comment :

- harmoniser le code C : les étudiants de toutes les filières doivent être capables de construire un code C de manière identique
- permettre une lecture rapide d'un code : les étudiants doivent être capables de lire un code rapidement, en détectant les déclarations, les blocs d'instruction et en comprenant les algorithmes utilisés
- favoriser la maintenance : les étudiants doivent être capables de faire évoluer très rapidement un code source, il doit donc être compréhensible et non obfusqué.

Organisation des projets

Un projet modélise une application complète et est constitué d'un ou de plusieurs fichiers. En général, le projet donne lieu à la création d'une arborescence de répertoires. Cette création est soumise à des règles d'organisation et il est nécessaire de classer les fichiers par type. Une organisation correcte devrait être :

/project_name	Répertoire racine du projet

/sources	Répertoire contenant tous les fichiers sources (il est possible de subdiviser en un répertoire pour les fichiers .c et un autre pour les fichiers .h)
/resources	Répertoire contenant les fichiers ressources utilisées par le projet (images, vidéos, fichiers XML, de données …)
/ext /library_name1 /include /lib /library_name2 /include /lib ….	Répertoire contenant les différentes librairies utilisées par le projet. Chacune d'entre elles est stockée dans son propre sous-répertoire, et possède ses propres fichiers organisés dans un répertoire include pour ses .h, et lib pour ses .a ou .lib (ou . quelconque)
/bin	Répertoire contenant l'exécutable du projet
/documentation	Répertoire contenant toute la documentation du projet
/INSTALL	Répertoire contenant tous les fichiers nécessaires à l'installation du projet sur une plate-forme.

Conventions de nommage

Les conventions de nommage indiquent comment définir le nom d'une variable, d'un type, d'une fonction ou de toute autre déclaration. Pour construire un code clair, un certain nombre de règles doivent être respectées :

- un identifiant doit être construit en anglais,
- un identifiant doit être explicite : price est préférable à p, counter à i,
- même si certains langages exotiques autorisent des smileys comme caractères possibles, il faut limiter les caractères utilisés à la liste suivante : lettres [a-zA-Z] sans caractères accentués, chiffres [0-9] ou dans certains cas le symbole _ (underscore)
- un identifiant commence par une lettre

Indépendamment des règles précédentes, se pose le problème de la casse et des identifiants formés de plusieurs mots. Les règles suivantes sont proposées :

fichiers/répertoire/projets	minuscules, sans espace, snake_case	student_management.c
constantes et macros du préprocesseur	DEF_CONST	#define PI 3.14 cont double TVA_RATE = 0.1;
variables	camelCase	char studentName[SIZE];
types définis par le développeur (structures, énumérations, unions, ...)	CamelCase, commence par une majuscule	struct Customer { … };
fonctions	camelCase()	void printStudent(Student);

Modèles d'organisation de fichier

Deux cas de figure :

- la construction d'un fichier de type header (fichier .h)

/* ** Filename : ** ** Made by : ** ** Description : */	Toujours commencer par une description commentée en anglais du fichier en cours
#include	Les directives du préprocesseur
typedef struct Student { … } Student;	Les types définis par le développeur
double computeAvg(Student student);	Les prototypes

- la construction d’un fichier de type source (fichier .c)

/* ** Filename : ** ** Made by : ** ** Description : */	Toujours commencer par une description commentée en anglais du fichier en cours
#include	Les directives du préprocesseur
double compute(double valueA, double value B) { double avg; avg = (valueA + valueB) / 2.; … return avg; }	Les fonctions
int main(int argc, char ** argv) { printf(“Hello ESGI world !”); return EXIT_SUCCESS; }	Un fichier.c peut également contenir une fonction main, pouvant retourner trois valeurs: ● 0 ou EXIT_SUCCESS en cas de réussite ● EXIT_FAILURE en cas d’échec

Indentation et formatage

La présentation d'un code source est fondamentale : elle doit être claire, lisible et surtout bien indentée. Il est absolument indispensable de respecter les règles suivantes :

- écrire une seule instruction par ligne
- une ligne ne doit pas dépasser 80 caractères
- un espace doit être saisi avant et après chaque opérateur (sauf pour les opérateurs unaires)

Bad	**Good**
in=in+1;	index = index + 1;
in ++;	index++;
equ=kid ==jid?0 :1;	equal = kId == jId ? 0 : 1;

- toutes les instructions doivent être correctement indentées, ce qui signifie que dès qu'il y a imbrication d'instructions, celles-ci doivent être décalées par rapport à la marge gauche. Une valeur de tabulation de 4 caractères doit être utilisée pour cela

Bad	**Good**
int main(int argc, char **argv) { int sum, val; val = 10; sum=15; sum=sum+val; if(sum<3) {printf("Hello");printf("%d",sum); } }	int main(int argc, char ** argv) { int sum; int value; value = 10; sum = 15; sum = sum + value; if(sum < 3) { printf("Hello"); printf("%d", sum); } }

Déclarations

Les variables sont toujours programmées en début de bloc. Il ne doit y avoir qu'un seule déclaration par ligne de code avec un espace unique entre le type et la variable

Bad	**Good**
int code;	int code;
int code,id,zip,value;	int code; int id; int zip; int value;
int code, id = 3;	int code; int id = 3;
int tab [10];	int array[10];
int array[] = {3,4,5}	int array[] = { 3, 4 , 5 };
double* ptrPrice;	double *ptrPrice;
double price = 12.3; double*ptrPrice=&price;	double price = 12.3; double *ptrPrice = &price;
char * * buffer;	char **buffer;

Formatage des instructions structurées

Ecrire convenablement les instructions structurées est fondamental. Les règles suivantes sont proposées :

- toutes les instructions structurées doivent être délimitées par des accolades (même si elles ne contiennent qu'une seule instruction)

- présentation du if :

Bad	**Good**
if(code==3) { printf(“3”); }	if (code == 3) { printf(“3”); }
if(code>0&&code < 10) { printf(“3”); }	if (code > 0 && code < 10) { printf(“3”); }
if (code == 3) { if(zip==4) if(id==7) { code=code+2;; } }	if (code == 3) { if (zip == 4) { if(id == 7) { code = code + 2; } } }
if (strcmp (answer, "yes")) code=1; else if (strcmp (answer, "no")) code =2; else if (strcmp (answer, "maybe")) code = 30; else code=0;	if (strcmp(answer, "yes")) { code = 1; } else if (strcmp (answer, "no")) { code = 2; } else if (strcmp (answer, "maybe")) { code = 30; } else { code = 0; }

- présentation du switch :

Bad	**Good**
switch(value) { case 1: code=2; y=5; f(akita);break; case 2: g(inu); break; }	switch(value) { case 1 : code = 2; zip = 5; f(akita); break; case 2 : g(inu); break; }

- présentation du for (respecter les bornes traditionnelles du C et partir de 0 à chaque fois que c'est nécessaire) :

Bad	**Good**
for (i = 0; i < counter; ++i) printf("Hello ESGI\n");	for (i = 0; i < counter; ++i) { printf("Hello ESGI\n"); }
for(i = 1; i <= n; i++);	for (i = 0; i <= n; i++) ;

- présentation du while :

Bad	**Good**
value = 0; while (value < counter) scanf("%d", &value);	value = 0; while (value < counter) { scanf("%d", &value); }
while(*dest++ = *src++);	while((*dest++ = *src++)) ;

- présentation du do while :

Bad	**Good**
do process(&counter, array); while (counter != 0);	do { process(&counter, array); } while (counter != 0);

Fonctions

Une fonction a vocation à modéliser un traitement unique, et ne peut servir à modéliser deux ou plusieurs tâches. Sa construction doit donc être de qualité; toujours la considérer comme une brique logicielle, qui une fois testée doit fonctionner dans toutes les circonstances possibles. (Une erreur typique serait donc de construire une fonction et de faire un affichage avant l'instruction return). Il est nécessaire de respecter les règles suivantes :

- Construire le prototype d'une fonction est obligatoire quelle que soit la position de la fonction dans un fichier source
- La taille d'une fonction est limitée, ne serait-ce que pour en permettre une maintenance aisée : le nombre d'instructions ne devra pas dépasser 30.

Bad	**Good**
void print() { int valA,valB,valC; valA=10; printf("%d",valA); }	void print(void); void print(void) { int valA; int valB; int valC; valA = 10; printf("%d", valA); } *// norme : depuis c89, void préférable* *// en C et interdit en C++*
int add(int, int); int add(int a, int b) { int c; c= a * b+10; printf("%d",c); return c; } void f() { int i=10, j=25, k = add(a,j); }	int add(int, int); void f(void); int add(int valA, int valB) { int result; result = valA * valB + 10; return result; } void f(void) { int numberOne = 10; int numberTwo = 25; int addresult; addResult = add(numberOne, numberTwo); printf("%d", addResult);

	}

- Décomposer le projet en bibliothèques de fonctions, ce qui permet leur réutilisabilité :
    - stocker les déclarations et les prototypes dans un fichier inclus .h
    - stocker les fonctions dans des fichiers .c
    - construire un fichier principal ne contenant que la fonction main
    - la taille d'un fichier source ne doit pas être gigantesque sous peine de difficultés à la maintenance : 500 lignes semblent raisonnables pour s'arrêter.

Structures et unions

Les structures et les unions permettent de regrouper plusieurs données à l'intérieur d'une seule variable. L'union permet d'accéder/modifier à une seule donnée à la fois (les anciennes valeurs étant écrasées), la structure permet d'accéder/modifier toutes les données.

Bad	**Good**
typedef struct Rectangle { int x;int y; int width;int height; } Rectangle;	typedef struct Rectangle { int x; int y; int width; int height; } Rectangle;
struct Circle { int x;int y;int radius; }; typedef struct Circle Circle;	typedef struct Circle { int x; int y; int radius; } Circle;
typedef union Shape { Rectangle rect; Circle circle; } Shape;	typedef union Shape { Rectangle rectangle; Circle circle; } Shape;

Commentaires

Il est indispensable à tout développeur de savoir commenter un code, sans sous-estimer ni surestimer la masse de texte à écrire.
Les principales règles à suivre sont les suivantes :

- ne jamais livrer un code non commenté, il ne sera pas compréhensible par le lecteur
- ne jamais livrer un code trop commenté, les instructions trop séparées les unes des autres ne seraient pas compréhensibles (un commentaire par instruction est une erreur grossière)
- commenter le code pendant sa construction (personne ne le fait une fois le développement terminé)
- commenter ce qui est difficile à comprendre, pas ce qui est compréhensible : l'idée est d'expliquer ce qui est fait (par exemple, recherche d'un élément dans un tableau) et pas comment (par exemple, dire qu'on utilise une boucle for, avec un indice variant entre 0 et 100, que l'on compare deux éléments etc… ne sert strictement à rien)
- construire un commentaire globale en début de code sources (rôle du programme, de l'API etc…), et un commentaire obligatoirement avant chaque fonction