Standard de programmation

Laurent Philippe

Résumé

Ce document est la référence du style de programmation. Il définit les règles qui doivent être appliquées pour l'écriture de "bon" code en C et d'autres langages. Le but de ces règles est de forcer le programmeur à concevoir et à écrire le code de manière plus lisible.

Introduction

Vous avez probablement déjà remarqué qu'il n'est pas toujours très facile de relire le code d'un autre, que certains codes sont plus "beaux" que d'autres. Cet esthétisme du code vient de sa simplicité et de sa facilité de lecture. Les standards de programmation ont été développés pour permettre une meilleure compréhension des programmes au moment de leur relecture, soit dans le cadre de la maintenance du code, soit dans le cadre de développements de logiciels à plusieurs. Leur but est de permettre une lecture plus aisée, donc plus rapide, en facilitant l'acquisition d'automatismes de lecture basés sur les repères définis par le standard. A vous donc d'essayer d'écrire du "beau" code... et de garder toujours à l'esprit qu'un jour quelqu'un (au moins votre prof) devra lire votre code et le comprendre.

1 Organisation des fichiers

Il n'existe pas de longueur maximale pour les fichiers de code, néanmoins il est raisonnable de limiter leur taille à un millier de lignes. Cela facilite la lecture, l'édition, etc. Les lignes ne doivent pas dépasser 80 colonnes car elles sont mal gérées par certains terminaux et par l'imprimante. Les lignes trop longues, du fait de l'indentation, sont souvent le symptôme d'une mauvaise structuration du programme.

1.1 Fichiers de code

Les fichiers ".c" contiennent le code : programme principal et fonctions. Ils sont organisés de la manière suivante :

1. une entête telle qu'elle est définie plus loin,
2. les inclusions de fichiers ".h", les fichiers du système tels que *stdio.h* doivent être inclus avant les fichiers de l'utilisateur,

3. les `define` et `typedef` qui s'appliquent à la totalité du fichier, dans l'ordre : les macros constantes, puis les macros fonctions, les `typedef` et les `enum`.
4. la définition des données globales, dans l'ordre : *extern*, données non-`static` et données `static`,
5. les fonctions classées par ordre d'importance.

Chaque fichier de code doit commencer par une entête générale qui a le format suivant :

```
/*
 ************************************************************
 *
 *  Programme : exempleStyleC.c
 *
 *  ecrit par : Laurent Philippe et ...
 *
 *  resume :    donne un exemple de programme ecrit avec le
 *              standard de programmation.
 *
 *  date :      22/05/95
 *
 ************************************************************
 */
```

Dans un même fichier, on regroupe les fonctions qui traitent d'une même partie du code ou, comme dans les bibliothèques, les fonctions qui réalisent un même type de traitement (affichage, traitement d'une chaîne de caractères, etc.).

1.2 Fichiers entête

Les fichiers ".h" contiennent des déclarations : principalement des types, classes ou des prototypes de fonctions. Ces fichiers contiennent les déclarations qui sont utilisées par plus d'un fichier. La séparation entre fichiers ".h" se fait sur les frontières logiques de séparation des données (déclarations se rapportant à la même partie du logiciel, données dépendantes d'une même machine, etc.). La définition de variables dans les fichiers entête est à éviter, on risque de définir plusieurs fois la même variable, en incluant plusieurs fois le même fichier.

Donnez le chemin d'accès aux fichiers entête, soit par rapport au répertoire local, soit dans les répertoires standards en utilisant la syntaxe <*name*>. Évitez de mettre le chemin absolu, cela peut entraîner des problèmes de portage.

Il peut être utile de mettre ces quelques lignes au début et à la fin de votre fichier ".h", pour éviter les doubles inclusions :

```
# ifndef EXAMPLE_H
#   define EXAMPLE_H
#   include VOTRE_INCLUDE
#   ifndef CHECK_DEFINE
#     include VOTRE_INCLUDE2
#   endif /* !CHECK_DEFINE */
... /* body of example.h */
#endif /* !EXAMPLE_H */
```

2 Les espacements

Ne soyez pas avares sur les espaces. Les espaces permettent d'améliorer la compréhension visuelle du code : ils permettent d'en souligner la structure. Les sauts de lignes peuvent, par exemple, aider à mieux comprendre les expressions complexes telle que :

```
c = (a == b)
    ? d + f(a)
    : f(b) - d;
```

2.1 Taille d'indentation

La taille d'indentation est le nombre d'espaces ajoutés à chaque nouveau bloc de code. Cette taille est fixée à **4**. Cela permet de donner une structure au code sans réduire trop la taille des lignes. Les lignes trop longues doivent être coupées et la fin de la ligne doit alors être suffisamment indentée pour rendre la coupure évidente, ne pas la confondre avec une instruction.

Les indentations peuvent être mises automatiquement par la commande `indent`. Il faut utiliser : `ident -i4 exempleStyleC.c`.

2.2 Les caractères blancs

Dans les fonctions
Dans un appel de fonction, il ne doit pas y avoir de blanc entre le nom de la fonction et la parenthèse gauche. Il doit y avoir un blanc après chaque virgule qui sépare un argument. Par exemple :

```
strncpy(dest, src, tail);
```

Au moins deux sauts de ligne doivent précéder la déclaration d'une fonction.

Autour des opérateurs
Il ne doit pas y avoir de blanc entre un opérateur unaire et son opérande :

```
~x      -x      x++     --x     &x     !x
++x     x++     *x
```

Par contre, dans le cas d'un *casting*, il doit y avoir un blanc entre le type et la variable :

```
surface = (float) input;          buffer = (int*) malloc(size);
```

Les opérateurs suivants doivent être encadrés de blancs :

```
x == y     x != y     x >= y     x <= y
x || y     x && y     x < y      x > y
x + y      x - y       x * y      x / y      x % y
x = y      x += y     x -= y     etc
x & y      x | y      x ^ y
x << y     x >> y
```

Les opérateurs binaires suivants ne sont pas séparés de leurs opérandes par des blancs, du fait de leur précédence :

```
x.y       x->y      x[y]
```

L'opérateur tertiaire `?` doit être entouré de blancs :

```
x ? y : z
```

Autour des séquences de contrôle

Les mots clef `if`, `while`, `for` et `switch` sont toujours séparés de la parenthèse gauche par un blanc. Ils ont donc la structure suivante :

```
if (expr) {
while (expr) {
for (expr1; expr2; expr3) {
switch (expr) {
```

Une accolade ouvrante doit être séparée de la parenthèse précédente par un blanc. Une accolade fermante doit être séparée de la suite par un blanc. Par exemple :

```
if (expr1) {

    instructions;

}
else {

    do {

        instructions;

    } while (expr2);
}
```

Dans les déclarations
Dans une liste de déclarations, le nom des variables et les commentaires doivent être alignés sur les mêmes colonnes :

```
int             maChauss;    /* descipteur de la chaussette    */
struct in_addr  chaussAdr;   /* adresse reseau de la chaussette */
struct servent  servAddr;    /* adresse du service */
```

3 Les commentaires

"Quand le code et les commentaires ne correspondent pas, les deux sont probablement faux"
Norm Schreyer

Un jour, quelqu'un devra lire votre code : pour le maintenir s'il a été écrit pour une entreprise ou pour le noter s'il a été écrit à l'Université. Cette personne devra comprendre ce que fait ce code, comment il le fait et pourquoi il le fait de cette manière, expliquez donc vos choix dans le code au moment où vous l'écrivez.

Les commentaires généraux doivent être placés dans l'entête des fonctions, ils décrivent ce que la fonction fait et comment elle le fait. Dans le texte on utilise deux structures de commentaires : les blocs de commentaires et les commentaires en ligne.

Les blocs de commentaires décrivent les algorithmes ou une partie du code. Ils sont de la forme suivante :

```
/*
 *  Je suis un bloc de commentaire
 *  Tu es un bloc de commentaire
 *  ...
 */
```

Les blocs de commentaires sont indentés de la même manière que la partie de code à laquelle ils font référence. Les séquences de début et fin de commentaire sont sur une ligne séparée et le texte est justifié à gauche.

Les commentaires "en ligne" doivent être courts. Ils sont généralement écrits au dessus de la ligne de code qu'ils décrivent. Par exemple :

```
if (argc > 1) {
    /* ouvre le fichier de la ligne commande */
    if (freopen(argv[1], ``r'', stdin) == FNULL) {
        perror (argv[1]);
    }
}
```

Des commentaires très courts peuvent apparaître sur une ligne de code. Dans ce cas, ils doivent être séparés du code par plusieurs espaces. Si plusieurs commentaires apparaissent sur des lignes consécutives, ils doivent commencer à la même colonne.

Utilisez sans retenue les deux types de commentaires mais gardez à l'esprit que le C n'est pas de l'assembleur : il vaut mieux définir des zones de commentaires qui font 3 à 10 lignes en expliquant ce qui est fait que de commenter chaque instruction ; la vision donnée par les commentaires est ainsi plus synthétique. Le taux de commentaires dans le programme doit être d'au moins 50%.

Pour mettre au point les programmes, et pour la bonne compréhension de leur fonctionnement, il est souvent utile de faire afficher des messages à l'écran, par exemple des messages d'erreur. Dans ce cas, les messages d'information (menus, traces, etc.) doivent être affichés sur la sortie standard alors que les messages d'erreur doivent apparaître sur l'erreur standard.

```
fprintf(stdout, "Tout roule pour le moment\n");
fprintf(stderr, "Rien ne va plus, j'arrete !!!\n");
```

4 Les déclarations

Il doit y avoir au plus une déclaration par ligne et chaque déclaration doit être accompagnée d'un commentaire. Les déclarations de variables globales doivent commencer en colonne 1. Toutes les déclarations de variables externes doivent être précédées du mot-clé `extern`, mais ce procédé doit être évité au maximum. La déclaration de variable globale doit être faite dans le fichier ".c". Dans la déclaration de pointeur, l'étoile doit toujours être attachée à la variable, pour être sûr qu'elle est de type pointeur.

Pour les structures et les unions, utilisez la forme suivante :

```
struct moto {
    int      cylind;          /* cylindree */
    TypMoto  type;            /* par ex: trail, custom, sport */
    char     marq[TAIL_MAX];  /* marque du vehicule */
}
```

Les types principaux, même s'ils sont entiers, doivent être définis avec un `typedef`. Pour définir un type énuméré, si les valeurs n'ont pas d'importance, il faut utiliser `enum`, avant la définition du type :

```
enum TypMoto{ TRAIL, CUSTOM, SPORT };
struct moto {
    int      cylind;          /* cylindree */
    TypMoto  type;            /* par ex: trail, custom, sport */
    char     marq[TAIL_MAX];  /* marque du vehicule */
}
```

Sinon utiliser `define` :

```
/*
 *  Definition pour bateau.type
 */
#define TRAIL  4
#define CUSTOM 5
#define SPORT  6
```

Cette définition doit se faire à proximité de la définition du type.

Essayez d'éviter au maximum les problèmes de types, ils cachent souvent des problèmes plus graves d'assimilation entre un pointeur et la valeur pointée. Pour typer correctement toutes vos données, typez vos constantes et évitez le typage par défaut de `define`. Par exemple, un pointeur nul sur un caractère doit être défini par :

```
    #define CNULL   (char*)NULL
```

pour être du bon type, la constante `NULL` étant de type entier.

Il doit toujours y avoir une (ou deux) lignes vides entre les déclarations et les instructions.

5 Conventions de nommage

Le choix d'un nom pour une variable, une fonction ou un type pose un dilemme : le nom doit être aussi explicite que possible sans être trop long. Les conventions de nommage suivantes vous aideront à trouver immédiatement à quoi correspond un nom.

Les variables à un caractère sont acceptables uniquement pour les indices de boucle.

```
for (i=0; i < SIZE; i++) {
```

Les noms de variables commencent toujours par une minuscule et ne doivent pas contenir de sous-tiret '_', qui est réservé aux variables systèmes. Si une variable est décrite par plusieurs mots, il suffit de mettre en majuscule la première lettre du second mot et des suivants. Un préfixe peut être ajouté devant chaque nom de variable pour traduire le groupe auquel elles appartiennent. Le nom des fonctions est attribué de la même manière que pour les variables. Par exemple :

```
clienCouran        clienAffichMoto()
```

Toutes les constantes (`enum` et `const`) et les macros numériques doivent être en majuscule. Éventuellement avec des sous-tirets comme séparateurs.

```
FAUX        GUZZI_CALIF        CONTRA_TIERS
```

Tous les types commencent par le préfixe **Typ**.

```
TypVoiture          TypMoto
```

6 Les fonctions

Pour structurer la vision du programme, on définit une entête par fonction. Le format standard d'une entête de fonction est le suivant :

```
/*
 * Fonction :    exempleFonctionC
 *
 * Parametres :  int paramEntier (parametre entier)
 *               char paramCara  (parametre caractere)
 *
 * Retour :      rien
 *
 * Description : cette fonction n'existe pas.
 *
 */
```

Attention cette entête doit être alignée en première indentation. Il va de soit que les différents champs de cette entête doivent être soigneusement remplis.

La déclaration de la fonction se fait sur une ligne et l'accolade ouvrante se trouve à sa fin. Si la fonction ne retourne rien, il faut lui donner le type `void`. De même, il ne faut pas utiliser le type `int` par défaut. Le nombre de paramètres est limité à 4 (éxécution plus rapide). L'accolade fermantes d'une fonction doit être sur une ligne séparée. Les instructions doivent être séparées des déclarations locales par une ligne vide. Les déclarations des variables doivent être sur une ligne et séparées de leurs affectations (aussi sur une ligne). Les fonctions ne doivent pas avoir plus de 25 lignes d'instructions (hors déclaration des variables). De plus, pour un fichier, il doit y avoir un maximum de 10 fonctions. Ainsi on respecte aisément la limite de ligne d'un fichier. Par exemple :

```
    /*
     * Fonction :    dernierClien
     *
     * Parametres :  TClien* listClien, liste de client (pointeur sur le début de la liste)
     *
     * Retour :      TClien*, pointeur sur le dernier client de la liste passée en paramètre
     *
     * Description : cherche le dernier element de la liste des
     *               clients, rend NULL s'il la liste est vide
     *               un pointeur sur le dernier sinon (pas une copie).
     *
     */

TClien* dernierClien(listClien){
    TClien *pClienCour;    /* pointeur sur le client courant */
    TClien *pDernClien;    /* pointeur sur le dernier client */

    if (listClien == NULL) {
        return (NULL);
    }

    for (pClienCour = listClien; pClienCour != NULL; ) {
        pDernClien = pClienCour;
        pClienCour = pClienCourant->suivant;
    }

    return (pDernClien);
}
```

7 Les instructions

Il ne doit y avoir qu'une seule instruction par ligne, à moins que les instructions soient en rapport :

```
    case GUZZI:  affichStruct(V_MOTOR);   break;
    case HONDA:  affichStruct(4_CYL);     break;
    case BMW:    affichStruct(FLAT_TWIN); break;
```

Dans ce cas, un espace doit être laissé entre deux instructions.

Pour un bloc de contrôle il faut utiliser la structure suivante :

```
control {
    instruction;
    instruction;
}
```

Dans le bloc, les instructions ont un niveau de plus d'indentation et elles sont toujours entourées d'accolades, même s'il n'y a qu'une seule ou pas d'instruction (dans ce cas il faut mettre un commentaire). Ceci permet d'ajouter facilement de nouvelles instructions et évite de nombreuses erreurs.

Il faut éviter les tests implicites. Par exemple, il ne faut pas écrire :

```
if (test)               if(func())
```

mais plutôt :

```
if (test == TRUE)       if(func() != FAIL)
```

C'est à dire que vous devez définir des constantes telles que `TRUE` ou `FAIL` pour faciliter la compréhension du code. De même, une valeur entière ne doit jamais être utilisée directement pour un test. Il faut la comparer à la valeur qui doit effectivement être testée (`if (tail)` doit être remplacé par `if (tail != 0)`). Dans une structure `case`, essayez de définir des constantes pour les différents cas, il est ainsi plus facile de savoir à quoi correspond chacun des cas.

La structure d'une instruction `if-else` est la suivante :

```
if (expr) {
    instructions;
}
else {
    instructions;
}
```

Ceci permet d'être sûr que le `if` est accompagné d'un *else*.

la structure d'une instruction `switch` est la suivante :

```
switch (expr) {
case V_GUZZI:
case V_DUCAT:
    instructions;
    break;
case BWM_3CYL:
    instructions;
    /* attention il n'y a pas de break */
case BWM_4CYL:
```

```
        instructions;
        break;
    default:
        instructions;
        break;
    }
```

Notons que les deux premiers cas ont le même traitement mais qu'ils sont sur des lignes différentes (ce n'est pas obligatoire s'il y a beaucoup de cas avec le même traitement). Un commentaire précise explicitement que le troisième cas n'a pas de `break`, pour éviter toute confusion.

Voilà, si vous respectez toutes ces règles, il ne fait pas de doute que votre code sera plus lisible, donc mieux compris et mieux noté !