ENSTA ParisTech
École nationale supérieure de techniques avancées

Petits conseils de programmation en C

Cours IN201
Systèmes d'exploitation

Marc Baudoin
<babafou+in201@babafou.eu.org>

Maîtriser un langage de programmation, ce n'est pas seulement en connaître la syntaxe. Il faut aussi savoir utiliser correctement les outils qu'il fournit et organiser son code source de manière intelligente et lisible. À cet égard, la programmation en C est une science tout autant qu'un art. Voici quelques petits conseils pour vous aider à développer vos qualités de programmation en C.

1 La lisibilité

Un programme en C est destiné à être compilé, c'est-à-dire transformé en un exécutable par un logiciel, le compilateur. Pour cela, il doit simplement respecter une certaine syntaxe, celle du langage C. La façon dont le code source est présenté (indentation des lignes, présence d'espaces, noms des variables) n'importe pas au compilateur. Ainsi, les deux programmes suivants, une fois compilés, généreront exactement le même exécutable :

```
#include<stdio.h>
#include<stdlib.h>
int main(int argc,char**argv)
{int i;for(i=0;i<10;i++){printf("i = %d\n",i);}exit(EXIT_SUCCESS);}
```

```
#include <stdio.h>
#include <stdlib.h>

int main ( int argc , char **argv )
{
   int i ;
```

```
7
8     for ( i = 0 ; i < 10 ; i ++ )
9     {
10       printf ( "i = %d\n" , i ) ;
11    }
12
13    exit ( EXIT_SUCCESS ) ;
14 }
```

L'exemple est évidemment poussé à l'extrême mais force est de reconnaître que le second programme, grâce aux espaces et à l'indentation, est bien plus lisible pour le lecteur humain.

Lorsqu'on programme, la lisibilité du code source est essentielle. En effet, elle permet :

- de saisir rapidement la structure du programme ;
- de faciliter la lecture et la compréhension du programme, par son auteur et par d'autres personnes.

En particulier, les erreurs sont bien plus faciles à détecter dans un programme lisible que dans un programme écrit sans souci de clarté.

La programmation est un art tout autant qu'une science. Un code source doit être agréable à lire avant toute chose.

Voici quelques règles de bon sens pour vous aider à écrire des programmes facilement lisibles.

1.1 L'indentation

L'indentation permet de faire clairement apparaître la structure d'un programme. Pour cela, l'idéal est d'écrire ses blocs ainsi :

```
instruction précédant le bloc
{
   déclaration des variables locales au bloc

   instruction1 ;
   instruction2 ;
   ...
}
```

Les accolades sont placées au même niveau d'indentation que le bloc précédent, l'une exactement sous l'autre, ce qui permet de repérer très facilement les accolades correspondantes.

Les accolades sont toujours présentes, même lorsqu'elles ne sont pas obligatoires. Ainsi, il vaut mieux écrire :

```
if ( i == 0 )
{
   j += 4 ;
}
```

plutôt que :

```
if ( i == 0 )
   j += 4 ;
```

De cette manière, le code source est plus cohérent (il y a des accolades partout), et on ne risque pas d'oublier de mettre les accolades si l'on rajoute des instructions dans le corps du test.

Chaque bloc est indenté de 3 espaces, ce qui permet de bien le différencier des blocs qui l'entourent. Il s'agit d'une valeur arbitraire mais qui n'est ni insuffisante ni excessive.

Chaque bloc est divisé en deux parties séparées par une ligne blanche : la déclaration des variables locales et les instructions.

> http://en.wikipedia.org/wiki/Indent_style

1.2 L'espacement et les lignes blanches

Il est important de mettre des espaces quasiment partout où cela est possible. Cela aère le code source et facilite grandement sa lecture en séparant visuellement les éléments de nature différente.

De même, il ne faut pas hésiter à insérer une ligne blanche pour séparer des parties du code source n'ayant pas de rapport direct entre elles.

1.3 80 caractères au plus par ligne

Tous les éditeurs de texte utilisent par défaut des lignes de 80 caractères de large. Afin que les autres personnes qui liront votre code source puissent en avoir la même vision, il est important d'en limiter la longueur à 80 caractères au plus par ligne. Les lignes trop longues sont de toute façon peu lisibles.

Par ailleurs, comme on peut couper une ligne partout ou l'on peut mettre une espace, il est parfois intéressant d'en profiter pour couper volontairement certaines lignes afin de rendre l'ensemble plus lisible. Considérons l'exemple suivant :

```
if ( x >= X_MIN && x <= X_MAX &&
     y >= Y_MIN && y <= Y_MAX )
{
   /* ... */
}
```

Le test a été réparti sur deux lignes, la première consacrée à la variable `x` et la seconde à la variable `y`. Compte tenu des similitudes entre les deux lignes, elles ont été alignées pour faire ressortir ces similitudes.

1.4 Le nommage des variables

Il faut donner aux variables des noms facilement compréhensibles. Par exemple, si, dans l'énoncé d'un projet de calcul numérique, les paramètres s'appellent v_0, α et x, il est cohérent d'appeler les variables correspondantes `v0`, `alpha` et `x`. En revanche, si aucun nom n'est précisé, il vaut mieux choisir des noms qui veulent dire quelque chose, même s'ils sont un peu longs (par exemple, `vitesse_initiale`, `angle` et `position` sont préférables à `v`, `a` et `p`).

1.5 Les commentaires

Utiliser des noms de variables facilement compréhensibles évite déjà d'avoir à mettre de nombreux commentaires triviaux.

Néanmoins, il est parfois indispensable de commenter certaines parties de code. Ces commentaires doivent apporter des informations supplémentaires et ne pas simplement paraphraser le code. Voici un exemple de commentaire totalement inutile :

```
i = 0 ;   /* on met 0 dans la variable i */
```

Voici un exemple de commentaire plus utile :

```
i = 0 ;   /* on reprend la recherche au début */
```

Il est également intéressant de faire précéder chaque déclaration de fonction d'un commentaire indiquant son but et, si nécessaire, la signification de ses arguments et de sa valeur de retour.

1.6 Inclure les fichiers d'en-tête dans l'ordre alphabétique

Hormis `sys/types.h`, qui doit toujours être inclus en premier, l'ordre dans lequel sont inclus les autres fichiers d'en-tête est sans importance.

Afin de les ordonner facilement, il est préférable de les inclure dans l'ordre alphabétique, ceux n'étant pas dans un répertoire particulier en premier, puis ceux étant dans un répertoire, dans l'ordre alphabétique des répertoires puis, pour un répertoire donné, dans l'ordre alphabétique des fichiers.

1.7 Rester simple

Il faut absolument éviter les instructions compliquées ou ambiguës. Ce n'est pas parce qu'on peut faire beaucoup de choses en une ligne qu'il faut en profiter dès que possible. Comment comprendre ceci ?

```
*p++ = ++q->i ;
```

Tout dépend des priorités des opérateurs, que personne ne connaît en totalité. Il est préférable de ne pas compter sur ces priorités et d'écrire son programme de manière explicite, quitte à la faire en plusieurs lignes si nécessaire. De toute façon, le compilateur transforme les opérations complexes comme celle-ci en une suite d'opérations plus simples. Ce n'est donc pas parce qu'on peut faire cinq opérations en une ligne que le programme sera plus rapide.

2 La programmation progressive

Le problème se pose souvent de savoir quand tester un programme. Faut-il écrire le programme en entier et le tester une fois terminé (avec en hors d'œuvre une farandole d'erreurs de compilation puis de plantages) ou écrire le programme petit à petit et le tester au fur et à mesure ?

La bonne méthode pour concevoir un programme est bien entendu la seconde. De manière plus détaillée, la démarche à suivre est la suivante :

1. écrire un petit morceau de programme qui se suffit à lui-même ;
2. compiler le programme (et donc corriger les erreurs de syntaxe) ;
3. tester le programme et vérifier s'il fonctionne correctement ; si ce n'est pas le cas, le corriger et répéter les étapes 2 et 3 jusqu'à obtention du résultat souhaité ;
4. reprendre à l'étape 1.

De cette façon, vous serez certain d'avoir, à tout moment, un programme qui fonctionne.

3 Les variables locales à un bloc

Le langage C permet de déclarer des variables globales (utilisables dans toutes les fonctions), locales à une fonction (utilisables uniquement dans cette fonction), mais aussi locales à un bloc. Ces variables n'existent que du début du bloc (son accolade ouvrante) à sa fin (son accolade fermante). Cette possibilité est très intéressante car elle permet de déclarer les variables au plus près de leur utilisation, d'en limiter la durée de vie et d'économiser de la mémoire :

```
if ( i == 0 )
{
   int tmp ;   /* variable locale au bloc */

   /* ici, on peut utiliser tmp */
}
/* maintenant, tmp n'existe plus */
```

Ainsi, la variable `tmp` n'est utilisable que dans le bloc de l'instruction **`if`**.

4 L'allocation dynamique de mémoire : les fonctions `malloc()`, `calloc()` et `free()`

Il est fréquent de ne pas savoir déterminer la taille d'un tableau avant l'exécution d'un programme. Par exemple, dans le cas d'un programme de multiplication de matrices, l'utilisateur peut fournir les dimensions des matrices à l'exécution. Le programme alloue alors la mémoire nécessaire au stockage des matrices, puis leurs éléments sont fournis par l'utilisateur, le programme les stocke dans les matrices, calcule leur produit, affiche le résultat et libère la mémoire occupée par les matrices.

L'allocation dynamique de mémoire se déroule toujours ainsi :

1. détermination de l'espace nécessaire ;
2. allocation de la mémoire ;
3. utilisation de cette mémoire ;
4. libération de la mémoire.

Deux fonctions permettent d'allouer dynamiquement de la mémoire : `malloc()` et `calloc()`. La fonction `free()` est utilisée pour libérer la mémoire allouée par ces deux fonctions.

> http://en.wikipedia.org/wiki/Malloc

4.1 La fonction `malloc()`

La fonction `malloc()` s'utilise ainsi. Pour allouer un objet de type `type`, on écrit :

```
type *ptr ;

ptr = ( type * ) malloc ( sizeof ( type ) ) ;
```

Il ne faut surtout pas, ainsi qu'on peut le voir dans certains programmes, déclarer soi-même la fonction `malloc()` comme retournant un pointeur vers `type`. En effet, supposons qu'on veuille allouer deux objets de types différents `type1` et `type2`. Si l'on déclare `malloc()` comme retournant un pointeur vers `type1` et qu'on alloue les deux pointeurs, l'allocation du second déclenchera une erreur :

```
type1 *malloc() ;

type1 *ptr1 ;
type2 *ptr2 ;

ptr1 = malloc ( sizeof ( type1 ) ) ;
ptr2 = malloc ( sizeof ( type2 ) ) ;   /* erreur */
```

La bonne façon de faire est d'inclure le fichier d'en-tête `stdlib.h`, qui déclare correctement `malloc()` comme retournant un pointeur de type **`void`** et d'utiliser une conversion explicite (*cast* en anglais) pour chaque type à allouer :

```c
#include <stdlib.h>

type1 *ptr1 ;
type2 *ptr2 ;

ptr1 = ( type1 * ) malloc ( sizeof ( type1 ) ) ;
ptr2 = ( type2 * ) malloc ( sizeof ( type2 ) ) ;
```

Après une allocation, il est important de vérifier si celle-ci s'est correctement déroulée :

```c
#include <stdlib.h>

type *ptr ;

ptr = ( type * ) malloc ( sizeof ( type ) ) ;
if ( ptr == NULL )
{
   /* la mémoire n'a pas pu être allouée */
}
```

4.2 La fonction `calloc()`

La fonction `calloc()` s'utilise de la même façon que `malloc()`, mais elle permet d'allouer plusieurs objets du même type :

```c
#include <stdlib.h>

int n = 10 ;
type *ptr ;

ptr = ( type * ) calloc ( n , sizeof ( type ) ) ;
```

Ici, le pointeur `ptr` pointe vers une zone de mémoire pouvant contenir 10 objets du type `type`, c'est-à-dire un tableau de 10 éléments. On peut alors utiliser `ptr` comme un tableau et écrire `ptr[0]`, `ptr[1]`...

L'allocation de tableaux à plusieurs dimensions se fait très simplement (on se limitera au cas d'un tableau bidimensionnel, mais la généralisation à d'autres tableaux est très simple).

Comme, en C, un tableau bidimensionnel est un tableau de tableaux, l'allocation d'un tableau de n lignes et m colonnes se fait en deux temps :

1. allocation d'un tableau de n pointeurs ;
2. allocation de n tableaux de m éléments.

```c
#include <stdlib.h>

/* allocation d'une matrice */

#define NB_LIG 4
#define NB_COL 3

double **ptr ;
int lig ;

ptr = ( double ** ) calloc ( NB_LIG , sizeof ( double * ) ) ;
for ( lig = 0 ; lig < NB_LIG ; lig ++ )
{
   ptr[lig] = ( double * ) calloc ( NB_COL , sizeof ( double ) ) ;
}
```

Comme pour `malloc()`, il est important de vérifier si l'allocation s'est correctement déroulée. Les vérifications ne figurent pas dans les exemples précédents afin de ne pas les alourdir.

4.3 La fonction `free()`

La fonction `free()` permet de libèrer la mémoire allouée par les fonctions `malloc()` et `calloc()`. Elle prend en argument le pointeur retourné par ces fonctions :

```c
#include <stdlib.h>

type *ptr ;

/* allocation de la mémoire */
ptr = ( type * ) malloc ( sizeof ( type ) ) ;
if ( ptr == NULL )
{
   /* la mémoire n'a pas pu être allouée */
}

/* utilisation de la mémoire allouée */

/* libération de la mémoire */
free ( ptr ) ;
```

5 De la bonne utilisation des comparaisons numériques

En raison de la représentation interne des données, il ne faut en aucun cas faire de comparaisons d'égalité (ou d'inégalité) entre nombres à virgule flottante (types **`float`** et **`double`**).

Supposons que, pour arrêter un calcul, une variable `erreur`, de type **`double`**, doive être nulle. Le programme contient donc une boucle avec un test de la forme :

```
do
{
   /* le calcul se trouve ici */
}
while ( erreur != 0.0 ) ;
```

En raison du cumul des erreurs d'arrondi (avec une variable de type **`double`**, on n'a « que » vingt chiffres significatifs), il se peut très bien que la variable `erreur` ne soit jamais exactement nulle !

La bonne façon de procéder est de faire un test d'infériorité de la valeur absolue de la différence de la variable et de la valeur qu'elle doit atteindre par rapport à un nombre suffisamment petit (*ouf !*). En clair, la condition pour qu'une variable v vaille une valeur v_0 est que :

$$|v - v_0| \leqslant \varepsilon$$

où ε est suffisamment petit (sa valeur est à fixer au cas par cas).

Le test ci-dessus devrait donc s'écrire :

```
#include <math.h>

do
{
   /* le calcul se trouve ici */
}
while ( fabs ( erreur )  >= 1.0e-10 ) ;
/* la fonction fabs() renvoie la valeur absolue de son argument */
```

> http://en.wikipedia.org/wiki/Floating_point

6 Conclusion

La programmation demande une grande rigueur. La lisibilité du code source est essentielle et la démarche de conception doit être progressive. Le langage C offre des outils permettant de créer des variables à très courte durée de vie et d'allouer dynamiquement de la mémoire. Enfin, lors des comparaisons numériques, il convient de prendre garde aux nombres à virgule flottante.

Pour finir sur une note philosophique, lorsqu'on programme, il faut toujours avoir à l'esprit ce principe fondamental :

> *L'ordinateur fait ce que vous lui dites de faire,*
> *pas ce que vous voulez qu'il fasse.*

Vous pouvez prendre un moment pour méditer...