REGLES POUR AMELIORER LA QUALITE DES PROGRAMMES EN LANGAGE C

J. LAPORTE[1], F. LEFEVRE[2]

[1] **CAP SESA TERTIARE**
[2] **E.D.F.**
Direction des Etudes et Recherches
EP/SDM
6 quai Watier
78400 CHATOU

Résumé: On définit un ensemble de règles destinées à améliorer la qualité du code C. Les principes décrits ici ont pour but l'amélioration de la lisibilité et de la portabilité (UNIX-MS DOS) des sources.

Mots-clés: règles de programmation, langage C, normes, portabilité des sources.

1. INTRODUCTION.

Ce document cherche à dégager de l'expérience acquise, ici ou là, les règles fondamentales pour améliorer la qualité des logiciels.

L'expérience étant, comme on le sait, la somme des erreurs commises, il ne prétend donner aucune leçon.

2. REGLES COMMUNES DE QUALITE DANS LA PROGRAMMATION.

2.1 La loi d'opacité maximale

On peut dégager une règle psychologique et sociologique, malheureusement, commune en matière de programmation : la loi d'opacité maximale.

C'est, en effet, une tendance naturelle du programmeur, que de mettre son orgueil professionnel à rendre le texte de ses programmes, le plus opaque possible.

Nous laisserons à d'autres le soin d'analyser les causes de ce défaut que nous nous bornerons à constater : le programmeur "génial" ("hacker" aux Etats-Unis), n'écrit pas son code comme le commun des mortels, ou, plus exactement, il n'écrit pas un code dans le but que le commun des mortels puisse facilement l'appréhender.

A l'inverse, c'est devenu un lieu commun que d'affirmer qu'un principe essentiel de la qualité des logiciels est la lisibilité des codes-sources.

Le code, en langage évolué ou non, n'est pas écrit seulement pour être exécuté sans erreur par la machine, mais peut-être surtout, pour être relu, compris, corrigé ou modifié un jour par un humain.

C'est pour ce lecteur-là, et non uniquement pour un compilateur, qu'on doit, dans une optique de qualité, rédiger un code.

Or, si le compilateur, fait son affaire des problèmes éventuels de syntaxe, il en va autrement de la clarté, du style qui sont des conditions importantes de lisibilité et dont le caractère sémantique résiste, aujourd'hui au moins, au contrôle automatique.

2.2 La qualité du logiciel.

Selon l'AFNOR, "la qualité d'un logiciel réside dans son aptitude à satisfaire les besoins de son utilisateur".

Certains auteurs ont cherché à isoler les critères d'une telle adéquation. En particulier pour BOEHM, un logiciel doit être :

- utilisable en l'état,
- maintenable,
- portable.

Ces critères se décomposent de la manière suivante :

portable		indépendant du matériel
utilisable en l'état	sûr	autosuffisant exact
	efficace	complet robuste
	commode	cohérent ergonomique autodescriptif structuré
maintenable	testable compréhensible modifiable	concis lisible extensible

Nous regrouperons ces critères autour des idées de lisibilité du code, d'une part, de portabilité d'autre part et de fiabilité, enfin.

2.3 La lisibilité du code.

Qu'est-ce que la lisibilité ?

Nous poserons comme principe qu'un code lisible est celui qui peut être compris aisément, dans un but opérationnel, par une tierce personne du même niveau de compétence que le ou les auteurs, dans un futur éloigné.

On peut dire, en outre, que la signification d'un programme doit se déduire facilement de son texte, sans usage excessif, voire exclusif de la documentation.

Ce concept de lisibilité implique donc une compréhension opérationnelle : la source d'un code exécutable doit être permanente alors même que le ou les auteurs se succèderont dans le temps et elle doit permettre d'acquérir aisément une connaissance permettant de la faire évoluer.

Le code-source concrétise à un instant donné une expertise, la lisibilité a donc comme but ultime de maintenir la compétence acquise.

Elle vise la pérennité des investissements logiciels, autant que leur modifiabilité en améliorant la communication entre programmeurs, à différentes époques du cycle de vie du logiciel.

Cet objectif implique également une compréhension globale alors que la source sera faite de strates différentes, témoins du passage de nombreux auteurs.

Il est donc nécessaire de faire en sorte, qu'existe, au sein d'un même projet, une certaine cohérence, sinon unicité, du style, de la disposition etc.

On donnera ci-dessous quelques principes permettant d'aller dans cette voie.

On aurait tort de ne les considérer que comme des règles "cosmétiques" : le problème de l'inflation des coûts de maintenance des grands programmes, largement dû à leur faible lisibilité, a pu être qualifié de " crise du logiciel".

Rappelons toutefois que ce problème, difficilement quantifiable, est comme tout ce qui est lié à la qualité industrielle, un problème d'état d'esprit.

2.4 Pour des codes-sources plus humains.

a) Taille humaine des modules.

La bonne compréhension opérationnelle impose de se référer à des modules relativement courts ; une centaine de lignes est, en général, considérée comme un maximum.

Au-delà, le lecteur sera contraint à refaire une analyse du problème (reverse engineering) [1].

b) Clarté contre performances.

L'auteur d'un programme est souvent imbu de performances.

Le code qu'il montre volontiers à ses collègues, utilise des astuces de programmation pour améliorer sa rapidité, par rapport à son nombre de lignes.

On fera plutôt confiance, dans ce but, aux qualités optimisatrices des compilateurs modernes ainsi qu'aux performances toujours améliorées du matériel.

L'échange de lisibilité et de clarté contre de la performance

[1] [MEYER/BAUDOIN 80] sont plus sévères puisque ils préconisent 10 noms d'objets et 20 lignes de code exécutable, comme une limite pour un lecteur moyen !

est toujours payant à terme.

c) Vocabulaire du programme.

On essaiera, autant que faire se peut, de choisir comme noms de variables ou de procédures un mot qui donne une claire indication sur le contenu où l'objet de l'entité en question :

Ainsi on évitera :

i,j,k
proc1,fonct1 etc.

pour utiliser la forme *verbe_nom* :

insere_fiche,
test_valeur_echantillon

On évitera, également, l'écriture suivante qui n'apporte rien :

for (indice_boucle=0;indice_boucle<MAX;indice_boucle++)

d) La Structuration.

Le concept de programmation structurée et les thèses de E.W. DIJKSTRA [2] sur "l'instruction *goto* considérée comme dangereuse" peuvent difficilement être ignorées.

Rappelons simplement les points suivants, en renvoyant le lecteur pour une étude approfondie, à [DIJKSTRA 68], [KNUTH 74], [MEYER et BAUDOIN 80] et [ARSAC 77] :

- la structuration améliore la lisibilité des programmes de façon incontestable,

- elle encourage la tendance à "faire simple" (*keep it simple*),

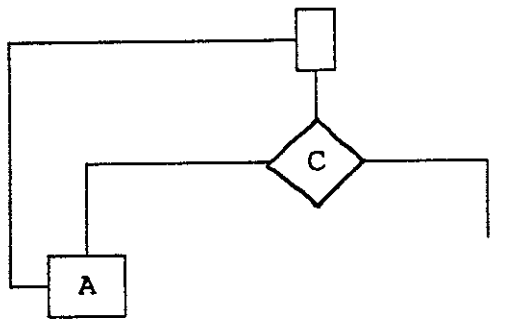
- tout algorithme, aussi complexe qu'il soit, peut être réécrit, en un programme n'utilisant que les structures de contrôle suivantes :

séquence et répétition *tant que*,

ou alternative et appel de fonctions récursives.

[2] L'école du Professeur DIJKSTRA (DAHL, HOARE, WIRTH, WARNIER ..) a développé une conception globale de la programmation, la considérant comme une des branches les plus difficiles des mathématiques appliquées. On consultera [MEYER/BAUDOIN 80, p.509], pour une vision complète de cette "conception ascétique" (op. cit) de la programmation.

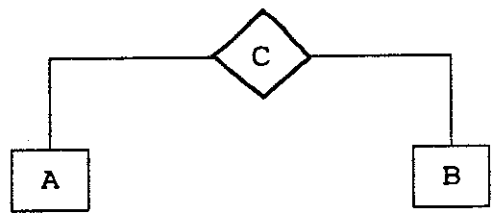
Ce théorème démontré par [BOEHM 66] rend vains les arguments du type : l'algorithme est trop complexe pour être exprimé sans *goto* [3]. Il existe, d'ailleurs, des langages déclaratifs sans instructions *goto*, par exemple PROLOG.



tant que C est vrai,
répéter le bloc A

```
while (C)
 {      }

do C
...
enddo
```



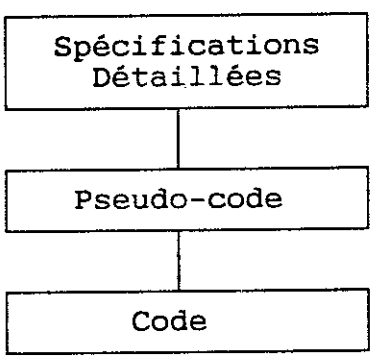

```
if (C)
   {      }
else
   {      }
```

- à un module correspondra un seul point d'entrée et de sortie,

- enfin, on limitera le nombre des variables ayant une sphère de définition globale (dans le temps et l'espace), COMMON en FORTRAN et variables globales en C.

Finalement et avec le recul, l'approche de la programmation structurée ne doit pas être un impératif dogmatique, mais plutôt une ligne de conduite dont on peut être sûr qu'elle améliore la qualité.

Cette approche sera encouragée par l'adoption du schéma de développement suivant :



[3] On s'accorde à reconnaître au *goto* une utilité dans les situations de gestion d'erreurs.

L'écriture du pseudo-code [4] n'est en effet possible qu'après un étude technique détaillée de l'algorithme ou des traitements à réaliser et évite le codage trop précoce.

L'expérience montre, en effet, que les codes très imbriqués (*spaghetti codes*) sont issus d'un développement au coup par coup (*patchwork*) sans effort d'analyse "terminal éteint".

Nous terminerons ce paragraphe par une constatation personnelle que l'on nous pardonnera d'ériger en règle : tout augmentation de l'effort d'analyse "terminal éteint" entraîne une diminution plus que proportionnelle du temps et de la pénibilité du codage.

e) La Présentation du code.

En C on n'utilisera qu'une instruction par ligne, en l'alignant avec les autres et en indentant les différents niveaux de { } (délimiteurs de blocs).

L'éditeur vi (il a quelques qualités) rapproche automatiquement le } des {.

Enfin, l'outil cb (C beautifier), permet d'imposer un style unique aux différents fichiers-sources.

```
cb < fichier_laid > fichier_beau
```

En FORTRAN 77, en l'absence d'outils automatiques, on est amené à poser des règles plus nombreuses (cf. normes MODULAD).

2.5 La transportabilité du code.

La transportabilité est la qualité intrinsèque d'un code liée à la facilité de le faire fonctionner sur des machines différentes.

Dans l'environnement UNIX, de même qu'en cas de migration DOS vers UNIX et vice versa, cette transportabilité n'existe qu'au niveau du code-source, ce qui implique, au moins, de satisfaire les conditions suivantes :

- une identité de langage de programmation,

- une compatibilité des modules ou fonctions extérieurs utilisés (*libraries*),

- une absence de référence au matériel.

[4] Cf. [MEYER/BAUDOIN 80] pour des modèles de pseudo-codes, par exemple, page 420.

Toutefois, l'expérience montre que deux problèmes supplémentaires peuvent survenir sur des codes portables aux sens des conditions précédentes.

- on doit avoir identité de résultats numériques sur les deux machines avec un même code compilé,

- on doit avoir une portabilité des données en entrée ou en sortie du même code-source.

On envisagera, ici exclusivement, le langage C.
Le C portable (UNIX System V, BSD 4.X, MS-DOS, OS/2).
Une normalisation du C est encore en projet à l'ANSI. Certains compilateurs, sous DOS (MSC, Turbo C), peuvent contrôler le respect de la norme.

Le C ANSI, fournira de nombreuses extensions parmi lesquelles on peut noter :

- la possibilité de définir des types énumérés (comme en Pascal),
- le passage de structures comme paramètres de fonctions, etc.

En outre, une nouvelle syntaxe pour les déclarations de fonctions, proche de celle du Pascal, permettra aux compilateurs un contrôle actif de la cohérence des types des paramètres d'appel.

Toutefois, dans l'attente de l'adoption et de la diffusion large de cette norme, le C recommandé comme commun dénominateur est celui de KERNIGHAN et RITCHIE, tel qu'il est présenté dans le manuel de référence officiel du langage (Appendice A, du manuel [K&R 78]).

En outre, certaines conditions minimales sont à respecter, pour qu'un code C puisse être transporté, d'un système à un autre.

1- pas de code assembleur (certains compilateurs comme Turbo C, permettent d'écrire du code assembleur à l'intérieur du C : code *inline*),

2- pas de code lié à une machine (ou architecture) particulière ; notamment faire attention aux problèmes liés à la conversion de types, aux comparaisons entre réels [5] etc.

[5] Pour comparer les deux flottants f1 et f2, on testera :
ABS(f1 - f2) < EPSILON (EPSILON variant d'une machine à l'autre) Cf [MEYER/BAUDOIN p. 76.

Si l'on ne peut éviter cela, on isolera les parties liées à un système particulier, dans un fichier inclus :

```
#include "vax.h"
```

et on utilisera au maximum le préprocesseur C, pour paramétrer les constantes.

```
#define  MAX  10000
#define  MIN  1
```

3- pas d'adresse absolue dans un programme :

```
char * p;
p = 0xb000;
```

L'instruction précédente qui fait, dans l'architecture IBM-PC, pointer p sur la mémoire vidéo pour y écrire, est une hérésie. Il faut utiliser à la place des fonctions spécialisées.

4- ne pas utiliser, les multiples idiotismes des différents C, qui ne sont pas standards au sens K&R :

```
int getch(c)
```

L'instruction précédente sur IBM-PC et en MSC ou Turbo C, retourne un caractère lu au vol, sans attente de retour de chariot.

De même, sur certaines stations de travail (HP 9000), les noms d'identificateurs peuvent avoir une longueur très importante (jusqu'à 255).

On respectera cependant, par prudence, le règle K&R des 8 caractères de longueur (Cf. [K&R] p.179).

5- ne pas utiliser indifféremment des entiers et des caractères , ce qui est autorisé par K&R mais peut poser des problèmes de cadrage, d'une machine à l'autre :

```
int i;

i = '@';
```

6- Point mal connu, l'ordre d'évaluation des paramètres d'une fonction n'est pas garanti s'effectuer de droite à gauche, bien que généralement cela soit le cas !

Se méfier donc de :

```
f(a, b, a=g(b));
```

qui ne donne pas le même résultat suivant le sens de l'évaluation.

Enfin, on notera d'expérience que la portabilité est quasi-parfaite de système UNIX à système UNIX.

System V <--> BERKELEY BSD 4.X

Se méfier toutefois des fonctions de bibliothèque HP-UX dont HP est l'auteur ; l'origine est généralement mentionnée dans le manuel et on pourra contrôler avec l'UNIX BSD 4.3, plus standard, de la station SUN4.

Le C sous XENIX (SCO ou Microsoft) est très proche du MSC de Microsoft, sous OS/2 ou DOS.

La dernière version du compilateur Microsoft est commune à DOS et OS/2, la compatibilité est très bonne avec le C UNIX, si l'on respecte les règles ci-dessus. Le dévermineur CODEVIEW est parfait.

QUICK C (Microsoft) et TURBO C (Borland), comportent de nombreuses fonctions non-standards spécifiques à l'IBM-PC (graphiques par exemple).

La meilleure approche UNIX-DOS, pour un produit nouveau, n'est pas celle du portage mais celle du développement croisé. Les modules sont développés sur PC, chargés sous UNIX grâce au réseau local, compilés et testés, dans les deux environnements.

2.6 La fiabilité du code.

Le troisième critère envisagé par BOEHM est celui de la fiabilité. Ce concept peut lui-même être décomposé en différentes conditions:

- cohérence : notations et terminologie sont uniformes dans le code source (Cf. problèmes des strates successives),

- robustesse : le code fonctionne correctement même s'il se trouve dans des conditions non conformes à celles prévues,

- traçabilité : il est possible de remonter du code vers les spécifications.

Nous rappellerons que pour arriver à satisfaire à ces critères de qualité, on ne saurait se contenter uniquement des tests.

La qualité du logiciel est obtenue par application des principes dégagés plus haut, dans l'ensemble du processus de fabrication (notion dite de qualité totale).

En outre, on doit garder à l'esprit ce qu'on a pu appeler la futilité des tests.

DIJKSTRA, toujours lui, a magistralement montré que les tests ne peuvent servir à prouver que la présence d'erreurs et non leur absence [6].

On en revient à l'idée de qualité totale, évoquée plus haut, puisque le programmeur ne peut compter de façon absolue, sur les tests pour palier ses éventuelles faiblesses.

3. REGLES D'UTILISATION OPTIMALES ET PRUDENTES DU C.

3.1 Documentation sur le C.

Le manuel pour s'initier ou travailler le Langage C est celui de Brian KERNIGHAN et Dennis RITCHIE, The C Programming Language, Editeur Prentice Hall, [K&R79].

On l'utilisera dans sa version anglaise, écrite dans un beau style concis, clair et pratique. La traduction française, éditée par Masson est médiocre.

Dans la suite de ce texte, on désignera le couple d'auteurs par leurs initiales K&R ; on parlera également de norme et de style K&R.

3.2 Points forts et faiblesses du C.

Le langage C est un langage puissant mis au point par RITCHIE, pour écrire un système d'exploitation (UNIX).

[6] "*Testing can show the presence, but never the absence of errors in software*", cité par [BUXTON 87]

Afin de caractériser la puissance et les faiblesses du C, nous emprunterons la plume de K&R eux-mêmes :

- "le langage C s'est avéré un langage plaisant, expressif et polyvalent",

- "il est souvent associé", à UNIX mais, "il n'est lié à aucune machine, à aucun système d'exploitation particulier",

- "il est indépendant de l'architecture d'une machine particulière et, avec un peu de soin, on peut écrire des programmes portables",

- "il peut être qualifié comme étant de bas niveau" ("*low level, it simply means than C deals with the same sort of objects that most computers do*"),

- "Néanmoins, il a prouvé qu'il était très efficace et très expressif dans une large gamme d'applications",

a) Réflexions sur le langage C.

Plaisant, le C l'est assurément, si l'on en juge par son succès auprès des développeurs.

Il peut l'être moins auprès des Chefs de projet, s'il est choisi et utilisé inconsidérément.

La mise au point d'un programme en C est certainement plus coûteuse qu'en langage Pascal [7], à cause notamment du fait que le langage est beaucoup moins fortement typé [8].

Ainsi, dans l'exemple suivant, le langage assimile une conditionnelle et une expression au sens de Pascal et autorise:

```
char *s="EDF";
int i=0;
while(*s++) i++;
```

pour parcourir la chaîne s et en calculer la longueur.

[7] Et fortement plus coûteuse qu'en ADA (rapport 1 à 5 selon certaines sources).
[8] "*not a strongly-typed language in the sense of Pascal or Algol 68*" [K&R p.3].

Pour la petite histoire et pour laisser s'exercer le lecteur, notons que :

```
while(*++s) i++;
```

ne donne pas le bon résultat.

Prudence donc, mais gardons à l'esprit que de telles constructions furent inventées pour créer UNIX et pas pour les besoins de la programmation scientifique.

On peut toujours, écrire le C clairement et prudemment, comme on le ferait en Pascal, en partant d'un pseudo code en français.

Ainsi on codera :

```
if (c==0)
```

au lieu de

```
if (!c)
```

beaucoup moins lisible.

b) Quelques fautes courantes en C.

Nous avons relevé ci-dessous, quelques fautes fréquentes chez les débutants, voire chez les autres. Elles montrent l'aspect permissif et donc dangereux du langage.

Mais tout a un coût et c'est là le prix à payer pour la puissance du C.

1- On veut tester l'égalité d'un entier et l'on écrit if (i=1) au lieu de if (i==1) ; le compilateur C, à bon droit, considère la chose comme excellente.

En effet, i=1 assigne la valeur 1 à l'entier i, évalué "vrai", dans la conditionnelle qui réussit toujours !

2- On écrit :

```
char * a,b;
```

pour déclarer en commun a et b comme des pointeurs vers des caractères, méconnaissant les règles de non-distributivité de l'opérateur *.

3- On écrit :

```
char *s;
s = malloc(10*sizeof(char));

s = "EDF";
```

pour allouer dynamiquement la place-mémoire pour 10 caractères.

Tout se gâte ensuite et l'on assigne le pointeur statique "EDF", à s (zone-mémoire statique de 3 octets seulement).

BIBLIOGRAPHIE

ARSAC *Nouvelles leçons sur la Programmation*, DUNOD, 1977.

BUXTON *The craft of Software Engineering*, ADDISON-WESLEY, 1987.

DIJKSTRA *The Humble programmer*, Turing Award Lecture, CACM 1972.

DIJKSTRA *A Discipine of Programming*, PRENTICE-HALL, ENGLEWOOD CLIFFS, 1976.

KERNIGHAN ET RITCHIE *The C Programming Language*, PRENTICE-HALL, ENGLEWOOD CLIFFS, 1979,

KERNIGHAN *Software Tools in Pascal*, ADDISON-WESLEY, 1976.

KNUTH *The Art of Computer Programming*, ADDISON-WESLEY, 1968,69,70.

MEYER B, BAUDOIN C *Méthodes de Programmation*, Collection EDF-DER, EYROLLES 1980.

MYERS G *Software Reliability, Principles and Practices*, WILEY 1976.

STORA et MONTAIGNE *La Qualité totale dans l'Entreprise*, Editions Organisation, PARIS 1987.

LEREDDE et BIGOT *MODULAD Normes de programmation FORTRAN* 77,Edition INRIA, Août 1989.