Guide Superflu de programmation en langage C

Matthieu Herrb

Version 2.3
Décembre 2006

Centre National de la Recherche Scientifique
Laboratoire d'Analyse et d'Architecture des Systèmes

Table des matières

Introduction

Ce document a pour but de rappeler certaines règles et techniques que tout le monde connaît pour développer une application de taille « sérieuse » en langage C.

Les audits menés sur des gros logiciels libres, en particulier, ceux menés par Theo De Raadt et les autres développeurs du projet OpenBSD [1] montrent que de nombreux programmeurs, même renommés, ont tendance à oublier ces règles, ce qui créée des bugs. Certaines de ces erreurs peuvent ouvrir une brèche dans la sécurité du système qui héberge le programme.

Les sources d'informations sur le sujet sont nombreuses et aucune n'a empêché les bugs connus d'exister ; c'est pourquoi je pense que ce document est superflu. En particulier, La Foire Aux Questions (FAQ) du forum Usenet `comp.lang.c` constitue une source d'information beaucoup plus riche, mais en anglais. Elle est accessible par l'URL :

`http://www.eskimo.com/~scs/C-faq/top.html`

Cette FAQ existe aussi sous forme de livre [1].

La bibliographie contient une liste d'autres articles ou ouvrages dont la lecture vous sera plus profitable. Si toutefois vous persistez à vouloir lire ce document, voici un aperçu de ce que vous y trouverez :

Le premier chapitre fait un tour parmi les problèmes les plus souvent rencontrés dans des programmes C.

Le deuxième chapitre donne quelques conseils sur le choix et le type d'algorithmes à utiliser pour éviter de réinventer la roue en permanence.

Le troisième chapitre s'intéresse à l'aspect sécurité informatique des algorithmes et de leurs implémentations, sachant que cet aspect doit être intégré au plus tôt dans l'écriture d'un programme.

Enfin, le quatrième chapitre traite du style du code source : indentation, choix des noms de variables, commentaires,...

Toutes les recomendations présentées ici ne sont pas des obligations à respecter à tout prix, mais suivre ces recommandations permettra de gagner du temps dans la mise au point d'une application et facilitera sa maintenance.

[1] `http://www.openbsd.org/`

Chapitre I

Quelques pièges du langage C

Le but de ce document n'est pas de faire un cours de langage C. Il y a des livres pour ça. Mais entre les bases du langage et la mise en œuvre concrète de ses fonctionnalités, il y a parfois quelques difficultés.

La plupart des erreurs décrites ici ne sont pas détectées à la compilation.

Certaines de ces erreurs conduisent systématiquement à un plantage du programme en cours d'exécution ou à un résultat faux, alors que d'autres conduisent à des situations que la norme du langage C définit comme « comportements indéterminés », c'est-à-dire que n'importe quoi peut se produire, selon les choix de programmation du compilateur ou du système.

Parfois, dans ce cas, le programme en question à l'air de fonctionner correctement. Une erreur ne se produira que dans certaines conditions, ou bien lorsque l'on tentera de porter le programme en question vers un autre type de machine.

I.1 Fautes de frappe fatales

Cette section commence par attirer l'attention du lecteur sur quelques erreurs d'inattention qui peuvent être commises lors de la saisie d'un programme et qui ne seront pas détectées par la compilation mais produiront nécessairement des erreurs plus ou moins faciles à détecter à l'exécution.

I.1.1 Mélange entre = et ==

Cette erreur est l'une des plus fréquentes, elle provient de la syntaxe du langage combinée à l'inattention du programmeur. Elle peut être très difficile à détecter. C'est pourquoi, il est indispensable d'avoir le problème à l'esprit en permanence.

= est l'opérateur d'affectation, alors que == est un opérateur de comparaison, mais en maths et dans d'autres languages de programmation on a l'habitude d'écrire $x = y$ pour désigner la comparaison.

Pour ceux qui ne verraient toujours pas de quoi il est question ici, `x = 0` est une expression valide en C qui affecte à `x` la valeur zéro et qui retourne la valeur affectée, c'est à dire zéro. Il est donc parfaitement légal d'écrire :

```
if (x = 0) {
    /* traitement à l'origine */
    ...
} else {
    /* traitement des autres valeurs */
    ...
}
```

Malheureusement le traitement particulier de $x = 0$ ne sera jamais appelé, et en plus dans le traitement des $x \neq 0$ la variable x vaudra 0 !

Il fallait évidemment écrire :

```
if (x == 0) {
    /* traitement à l'origine */
    ...
} else {
    /* traitement des autres valeurs */
    ...
}
```

Certains suggèrent d'utiliser systématiquement la construction suivante, qui a le mérite de provoquer une erreur à la compilation si l'on oublie l'un des « = » :

```
if (0 == x)
```

.

Cependant, ce problème n'est pas limité au cas de la comparaison avec une constante. Il se pose aussi lorsque l'on veut comparer deux variables.

I.1.2 Tableaux à plusieurs dimensions

En C les indices d'un tableau à plusieurs dimensions s'écrivent avec autant de paires de crochets qu'il y a d'indices. Par exemple pour une matrice à deux dimensions on écrit :

```
double mat[4][4];

x = mat[i][j];
```

Le risque d'erreur provient du fait que la notation `mat[i,j]` (qui est employée dans d'autres langages) est également une expression valide en langage C.

I.1.3 Oubli du break dans les switch

N'oubliez pas le break à la fin de chaque case dans une instruction switch. Si le break est absent, l'exécution se poursuit dans le case suivant, ce qui n'est en général pas le comportement voulu. Par exemple :

```
void
print_chiffre(int i)
{
  switch (i) {
     case 1:
       printf("un");
     case 2:
       printf("deux");
     case 3:
       printf("trois");
     case 4:
       printf("quatre");
     case 5:
       printf("cinq");
     case 6:
       printf("six");
     case 7:
       printf("sept");
     case 8:
       printf("huit");
     case 9:
       printf("neuf");
   }
}
```

Dans cette fonction tous les break ont été oubliés. Par conséquent, l'appel `print_chiffre(7);` affichera :

```
septhuitneuf
```

Ce qui n'est peut-être pas le résultat escompté.

I.1.4 Passage des paramètres par adresse

En langage C, les paramètres des fonctions sont toujours passés par valeur : il sont copiés localement dans la fonction. Ainsi, une modification d'un paramètre dans la fonction reste localisée à cette fonction, et la variable de l'appelant n'est pas modifiée.

Pour pouvoir modifier une variable de la fonction appelante, il faut réaliser un passage par adresse explicite. Par exemple, une fonction qui permute deux nombres réels aura le prototype :

```
void swapf(float *x, float *y);
```

Et pour permuter les deux nombres x et y, on écrira :

```
swapf(&x, &y);
```

Si on a oublié d'inclure le prototype de la fonction swap() avant de l'appeler, le risque est grand d'oublier de passer les adresses des variables. C'est une erreur fréquement commise avec la fonction scanf() et ses variantes.

I.2 Problèmes de calcul sur les nombres réels

Avant d'attaquer un programme quelconque utilisant des nombres réels, il est indispensable d'avoir pris conscience des problèmes fondamentaux induits par la représentation approchée des nombres réels sur toute machine informatique [2].

Dans de nombreux cas, la prise en compte de ces difficultés se fait simplement, mais il peut s'avérer nécessaire d'avoir recours à des algorithmes relativement lourds, dans le cas par exemple ou la précision de la représentation des réels par la machine n'est pas suffisante [3].

I.2.1 Égalité de réels

Sauf coup de chance, l'égalité parfaite ne peut être obtenue dans le monde réel, il faut donc toujours tester l'égalité à ϵ près. Mais il faut faire attention de choisir un ϵ qui soit en rapport avec les valeurs à tester.

N'utilisez pas :

```
double a, b;
...
if(a == b) /* Faux */
```

Mais quelque-chose du genre :

```
#include <math.h>

if(fabs(a - b) <= epsilon * fabs(a))
```

qui permet un choix de `epsilon` indépendant de l'ordre de grandeur des valeurs à comparer (À condition que `epislon` soit strictement positif).

I.2.2 Problèmes d'arrondis

La bibliothèque standard C propose des fonctions pour convertir des nombres réels en entiers. floor() arrondit à l'entier immédiatement inférieur, ceil() arrondit à l'entier immédiatement supérieur. Ces fonctions comportent deux pièges :

- elles retournent un type double. Il ne faut pas oublier de convertir explicitement leur valeur en type int lorsqu'il n'y a pas de conversion implicite.
- dans le cas d'un argument négatif, elles ne retournent peut-être pas la valeur attendue : `floor(-2.5) == -3` et `ceil(-2.5) == -2`.

La conversion automatique des types réels en entiers retourne quant à elle l'entier immédiatement inférieur en valeur absolue : `(int)-2.3 == -2`.

Pour obtenir un arrondi à l'entier le plus proche on peut utiliser la macro suivante :

```
#define round(x) (int)((x)>0?(x)+0.5:(x)-0.5)
```

I.2.3 Absence de déclaration des fonctions retournant des doubles

Le type double occupe sur la plupart des machines une taille plus importante qu'un int. Comme les fonctions dont le type n'est pas déclaré explicitement sont considérées comme retournant un int, il y aura problème si la valeur retournée était en réalité un double : les octets supplémentaires seront perdus.

Cette remarque vaut pour deux types de fonctions :
- les fonctions système retournant des doubles. L'immense majorité de ces fonctions appartiennent à la bibliothèque mathématique et sont déclarées dans le fichier `math.h`. Une exception à noter est la fonction strtod() définie dans `stdlib.h`.
- les fonctions des programmes utilisateur. Normalement toutes les fonctions doivent être déclarées avant d'être utilisées. Mais cette déclaration n'est pas rendue obligatoire par le compilateur. Dans le cas de fonctions retournant un type plus grand qu'un int, c'est indispensable. Utilisez des fichiers d'en-tête (`.h`) pour déclarer le type de *toutes* vos fonctions.

I.3 Style des déclarations de fonctions

L'existence de deux formes différentes pour la déclaration des paramètres des fonctions est source de problèmes difficiles à trouver.

Style K&R

En C « classique » (également appelé Kernigan et Ritchie ou K&R pour faire plus court), une fonction se déclare sous la forme [4] :

```
int
fonction(a)
  int a;
{
  /* corps de la fonction */
  ...
}
```

Et la seule déclaration possible d'une fonction avant son utilisation est celle du type retourné sous la forme :

```
int fonction();
```

Dans ce cas, tous les paramètres formels de types entiers plus petits que long int sont promus en long int et tous les paramètres formels de types réels plus petit que double sont promus en double Avant l'appel d'une fonction, les conversions suivantes sont effectuées sur les paramètres réels[1] :
- les types entiers (char, short, int) sont convertis en long int
- les types réels (float) sont convertis en double

Style ANSI

La norme ANSI concernant le langage C a introduit une nouvelle forme de déclaration des fonctions [5] :

```
int
fonction(int a)
{
```

[1] ici « réel » s'applique à paramètre, en opposition à « formel » et non à « type » (en opposition à « entier »)

```
  /* corps de la fonction */
  ...
}
```

Avec la possibilité de déclarer le prototype complet de la fonction sous la forme :

```
int fonction(int a);
```

Si on utilise ce type de déclaration, aucune promotion des paramètres n'est effectuée dans la fonction. De même, si un prototype ANSI de la fonction apparaît avant son appel, les conversions de types effectuées convertiront les paramètres réels vers les types déclarés dans le prototype.

Mélange des styles

Si aucun prototype ANSI d'une fonction (de la forme `int fonction(int a)`) n'a été vu avant son utilisation, le compilateur peut (selon les options de compilation) effectuer automatiquement les conversions de type citées plus haut, alors qu'une fonction déclarée selon la convention ANSI attend les paramètres avec le type exact qui apparaît dans la déclaration.

Si on mélange les prototypes ANSI et les déclarations de fonctions sous forme K&R, il est très facile de produire des programmes incorrects dès que le type des paramètres est char, short ou float.

I.4 Variables non initialisées

Les variables déclarées à l'intérieur des fonctions (« automatiques ») sont allouées sur la pile d'exécution du langage et ne sont pas initialisées.

Par contre les variables déclarées statiques sont garanties initialisées à zéro.

I.5 Ordre d'évaluation indéfini

Sauf exceptions, le C ne définit pas l'ordre d'évaluation des éléments de même précédence dans une expression. Pire que ça, la norme ANSI dit explicitement que le résultat d'une instruction qui dépend de l'ordre d'évaluation n'est pas défini si cet ordre n'est pas défini.

Ainsi, l'effet de l'instruction suivante n'est pas défini : `a[i] = i++;`

Voici un autre exemple de code dont le comportement n'est pas défini :

```
int i = 3;
printf("%d\n", i++ * i++);
```

Chaque compilateur peut donner n'importe quel résultat, même le plus inattendu dans ces cas. Ce genre de construction doit donc être banni.

Les parenthèses ne permettent pas toujours de forcer un ordre d'évaluation total. Dans ce cas, il faut avoir recours à des variables temporaires.

L'exception la plus importante à cette règle concerne les opérateurs logiques `&&` et `||`. Non seulement l'ordre d'évaluation est garanti, mais en plus l'évaluation est arrêtée dès que l'on a rencontré un élément qui fixe définitivement la valeur de l'expression : faux pour `&&` ou vrai pour `||`.

I.6 Allocation dynamique de la mémoire

Un des mécanismes les plus riches du langage C est la possibilité d'utiliser des pointeurs qui, combinée avec les fonctions malloc() et free() ouvre les portes de l'allocation dynamique de la mémoire.

Mais en raison de la puissance d'expression du langage et du peu de vérifications réalisées par le compilateur, de nombreuses erreurs sont possibles.

I.6.1 Référence à une zone mémoire non allouée

La valeur d'un pointeur désigne l'adresse de la zone mémoire vers laquelle il pointe. Si cette adresse ne correspond pas à une zone de mémoire utilisable par le programme en cours, une erreur (*segmentation fault*) se produit à l'exécution du programme. Mais, même si l'adresse est valide et ne produit pas d'erreur, il faut s'assurer que la valeur du pointeur correspond à une zone allouée correctement (avec malloc(), ou sous forme statique par une déclaration de tableau) par le programme.

L'exemple le plus fréquent consiste à référencer le pointeur NULL, qui par construction ne pointe vers aucune adresse valable.

Voici un autre exemple de code invalide :

```
int *iptr;

 *iptr = 1234;
printf("valeur : %d\n", *iptr);
```

`iptr` n'est pas initialisé et l'affectation `*iptr = 1234;` ne l'initialise pas mais écrit 1234 à une adresse aléatoire.

I.6.2 Référence à une zone mémoire libérée

À partir du moment où une zone mémoire a été libérée par free(), il est interdit d'adresser son contenu. Si cela se produit, on ne peut pas prédire le comportement du programme.

Cette erreur est fréquente dans quelques cas courants. Le plus classique est la libération des éléments d'une liste chaînée. L'exemple suivant n'est PAS correct :

```
typedef struct LISTE {
    int valeur;
    struct LISTE *suivant;
} LISTE;

void
libliste(LISTE *l)
{
    for (; l != NULL; l = l->suivant) {
        free(l);
    } /* for */
}
```

En effet la boucle for exécute `l = l->suivant` *après* la libération de la zone pointée par l. Or `l->suivant` référence le contenu de cette zone qui vient d'être libérée.

Une version correcte de `libliste()` est :

```
void
libliste(LISTE *l)
{
    LISTE *suivant;

    for (; l != NULL; l = suivant) {
        suivant = l->next;
        free(l);
    } /* for */
}
```

I.6.3 Libération d'une zone invalide

L'appel de free() avec en argument un pointeur vers une zone non allouée, parce que le pointeur est initialisée vers une telle zone, (cf I.6.1) ou parce que la zone a déjà été libérée (cf I.6.2) est une erreur.

Là aussi le comportement du programme est indéterminé.

I.6.4 Fuites

On dit qu'il y a fuite de mémoire lorsqu'un bloc alloué par malloc n'est plus référencé par aucun pointeur, et qu'il ne peut donc plus être libéré. Par exemple, la fonction suivante, censée permuter le contenu de deux blocs mémoire, fuit : elle perd le pointeur sur la zone tampon utilisée, sans la libérer.

```
void
mempermute(void *p1, void *p2, size_t length)
{
    void *tmp = malloc(length);
    memcpy(tmp, p1);
    memcpy(p1,p2);
    memcpy(p2,tmp);
}
```

En plus cette fonction ne teste pas le résultat de malloc(). Si cette dernière fonction retournait NULL, on aurait d'abord une erreur de référence vers une zone invalide.

Pour corriger cette fonction, il suffit d'ajouter `free(tmp);` à la fin du code. Mais dans des cas réels, garder la trace des blocs mémoire utilisés, pour pouvoir les libérer n'est pas toujours aussi simple.

I.7 Chaînes de caractères

Les chaînes de caractères sont gérées par l'intermédiaire des pointeurs vers le type char. Une particularité syntaxique permet d'initialiser un pointeur vers une chaîne constante en zone de

mémoire statique. Toutes les erreurs liées à l'allocation mémoire dynamique peuvent se produire plus quelques autres :

I.7.1 Débordement d'une chaîne de caractères

Cela se produit principalement avec les fonctions telles que gets(), strcpy(), strcat() ou sprintf() qui ne connaissent pas la taille de la zone destination. Si les données à écrire débordent de cette zone, le comportement du programme est indéterminé.

Ces quatre fonctions sont à éviter au maximum. La pire de toutes est gets() car il n'y a *aucun* moyen d'empêcher l'utilisateur du programme de saisir une chaîne plus longue que la zone allouée en entrée de la fonction.

Il existe des fonctions alternatives, à utiliser à la place :

- fgets() remplace gets()
- strlcpy() remplace strcpy()
- strlcat() remplace strcat()
- snprintf() remplace sprintf(). Malheureusement cette fonction n'est pas disponible sur tous les systèmes. Mais il existe un certain nombre d'implémentation « domaine public » de snprintf().

Exemple :

Le programme suivant n'est pas correct :

```
char buf[20];

gets(buf);
if (strcmp(buf, "quit") == 0) {
    exit(0);
}
```

Utilisez plutôt :

```
char buf[20];

fgets(buf, sizeof(buf), stdin);
if (strcmp(buf, "quit") == 0) {
    exit(0);
}
```

Il est à noter que les fonctions strncat() et strncpy() souvent présentées comme remplacement sûrs de strcat() et strcpy() ne le sont pas tant que ça. Ces fonctions tronquent les arguments qu'elles copient sans forcément insérer un caractère NUL pour marquer la fin du résultat.

strlcpy() et strlcat() ont été introduites pour corriger ce défaut. Ces fonctions terminent toujours la chaîne résultat par un NUL.

I.7.2 Écriture dans une chaîne en mémoire statique

La plupart des compilateurs et des éditeurs de liens modernes stockent les chaînes de caractères initialisées lors de la compilation avec des constructions du genre :

```
char  *s = "ceci est une chaîne constante\n";
```

dans une zone mémoire non-modifiable. Cela signifie que la fonction suivante (par exemple) provoquera une erreur à l'exécution sur certaines machines :

```
s[0] = toupper(s[0]);
```

L'utilisation du mot-clé const permet de détecter cette erreur à la compilation :

```
const char *s = "ceci est une chaîne constante\n";
```

I.8 Pointeurs et tableaux

Une autre puissance d'expression du langage C provient de la possibilité d'assimiler pointeurs et tableaux dans certains cas, notamment lors du passage des paramètres aux fonctions.

Mais il arrive que cette facilité provoque des erreurs.

I.8.1 Assimilation d'un pointeur et d'un tableau statique

Il arrive même aux programmeurs expérimentés d'oublier que l'équivalence entre pointeurs et tableaux n'est pas universelle.

Par exemple, il y a une différence importante entre les deux déclarations suivantes :
`char tableau[] = "ceci est une chaine";`
`char *pointeur = "ceci est une chaine";`

Dans le premier cas, on alloue un seul objet, un tableau de 20 caractères et le symbole `tableau` désigne directement le premier caractère.

Dans le second cas, une variable de type pointeur nommée `pointeur` est allouée d'abord, puis une chaîne constante de 20 caractères et l'adresse de cette chaîne est stockée dans la variable `pointeur`.

I.8.2 Appel de free() sur un tableau

Un tableau est une zone mémoire allouée soit statiquement à la compilation pour les variables globales, soit automatiquement sur la pile pour les variables locales des fonctions. Comme l'accès à ses éléments se fait de manière qui ressemble beaucoup à l'accès aux éléments d'une zone de mémoire allouée dynamiquement avec malloc(), on peut les confondre au moment de rendre la mémoire au système et appeler par erreur free() avec un tableau en paramètre.

Si les prototypes de free() et malloc() sont bien inclus dans la portée de la fonction en cours, cette erreur doit au minimum provoquer un warning à la compilation.

I.9 Entrées/sorties standard

La bibliothèque de gestion des entrées et sorties standard du langage C a été conçue en même temps que les premières versions du langage. Depuis la nécessité de conserver la compatibilité avec les premières versions de cette bibliothèque ont laissé subsister un certain nombre de sources d'erreur potentielles.

I.9.1 Contrôle des paramètres de printf et scanf

Les fonctions printf() et scanf() ainsi que leurs dérivées (fprintf(), fscanf(), sprintf(), sscanf(), etc.) acceptent un nombre variable de paramètres de types différents. C'est la chaîne de format qui indique lors de l'exécution le nombre et le type exact des paramètres. Le compilateur ne peut donc pas faire de vérifications. Ainsi, ces fonctions auront un comportement non prévisible si :

- le nombre de paramètres passé est inférieur au nombre de spécifications de conversion (introduites par %) dans la chaîne de format,
- le type d'un paramètre ne correspond pas au type indiqué par la spécification de conversion correspondante,
- la taille d'un paramètre est inférieure à la taille attendue par la spécification de conversion correspondante.

Certains compilateurs (dont `gcc`) appliquent un traitement particulier à ces fonctions et vérifient le type des paramètres lorsque le format est une chaîne constante qui peut être analysée à la compilation.

Rappelons les éléments les plus fréquemment rencontrés :

- les paramètres de scanf() doivent être les adresses des variables à lire, pas les variables elles-mêmes. Il faut écrire :
  `scanf("%d", &n);`
  et non :
  `scanf("%d", n);`
- pour lire un double avec scanf(), il faut utiliser la spécification de format `%lf`, par contre pour l'afficher avec printf() `%f` suffit.
  L'explication de cette subtilité se trouve à la section I.3. À vous de la trouver.

I.9.2 Lecture de chaînes de caractères

La lecture et l'analyse de texte formant des chaînes de caractères est un problème souvent mal résolu. Le cadre théorique général pour réaliser cela correctement est celui de l'analyse lexicographique et syntaxique du texte, et des outils existent pour produire automatiquement les fonctions réalisant ces analyses.

Néanmoins, dans beaucoup de cas on peut se contenter de solutions plus simples en utilisant les fonctions scanf(), fgets() et getchar().

Malheureusement ces fonctions présentent quelques subtilités qui rendent leur usage problématique.

- `scanf("%s", s);` lit un mot de l'entrée standard, séparé par des espaces, tabulations ou retours à la ligne. Cette fonction saute les séparateurs trouvés à la position courante jusqu'à trouver un mot et s'arrête sur le premier séparateur trouvé après le mot. En particulier si le séparateur est un retour à la ligne, il reste dans le tampon d'entrée.
- `gets(s)` lit une ligne complète, y compris le retour à la ligne final.
- `c = getchar();` et `scanf("%c", &c);` lisent les caractères un à un. La seul différence entre les deux est leur manière de retourner les erreurs en fin de fichier.

Le mélange de ces trois fonctions peut produire des résultats inattendus. En particulier appel à getchar() ou gets() après

```
scanf("%s", s);
```

retourneront toujours comme premier caractère le séparateur qui a terminé le mot lu par scanf(). Si ce séparateur est un retour chariot, gets() retournera une ligne vide.

Pour lire des textes comportant des blancs et des retours à la ligne, utilisez exclusivement fgets() ou getchar().

L'utilisation de scanf() avec le format `%s` est à réserver à la lecture de fichiers structurés simples comportant des mots-clés séparés par des espaces, des tabulations ou des retours à la ligne.

I.9.3 Lecture de données binaires

Les fonctions fread() et fwrite() de la bibliothèque des entrées/sorties standard permettent de lire et d'écrire des données binaires directement, sans les coder en caractères affichables.

Les fichiers de données produits ainsi sont plus compacts et plus rapides à lire, mais les risques d'erreur sont importants.

- Le rédacteur et le lecteur doivent être absolument d'accord sur la représentation des types de données de base ainsi écrites.
- Il est fortement conseillé de prévoir une signature du fichier permettant de vérifier qu'il respecte bien le format attendu par l'application qui va le lire.

I.10 Gestion des signaux

Un signal peut interrompre l'exécution d'un programme à n'importe quel moment. On ne connait donc pas l'état des variables lors de l'exécution de la routine de traitement du signal.

Par conséquent, seules des fonctions réentrantes peuvent être utilisées dans un gestionnaire de signal. Chaque système documente normalement la liste de ces fonctions autorisées dans les gestionnaires de signaux. Appeler toute autre fonction représente un problème potentiel.

Il est a noter que les fonctions de manipulation des entrées/sorties standard du C (printf(), etc.) ne sont pas réentrantes. Appeler l'une de ces fonctions est donc interdit, bien que pratique courante ; tous les programmes qui ne respectent pas cette règle courrent le risque de plantages plus ou moins aléatoires et plus ou moins fréquents à cause de cette violation des règles.

Parmi les autres fonctions interdites, notons malloc() et exit(). Utiliser _exit() à la place de cette dernière.

Voici la liste des fonctions autorisées sur le système OpenBSD :

- Fonctions standard

_exit()	access()	alarm()	cfgetispeed()	cfgetospeed()
cfsetispeed()	cfsetospeed()	chdir()	chmod()	chown()
close()	creat()	dup()	dup2()	execle()
execve()	fcntl()	fork()	fpathconf()	fstat()
fsync()	getegid()	geteuid()	getgid()	getgroups()
getpgrp()	getpid()	getppid()	getuid()	kill()
link()	lseek()	mkdir()	mkfifo()	open()
pathconf()	pause()	pipe()	raise()	read()
rename()	rmdir()	setgid()	setpgid()	setsid()
setuid()	sigaction()	sigaddset()	sigdelset()	sigemptyset()
sigfillset()	sigismember()	signal()	sigpending()	sigprocmask()
sigsuspend()	sleep()	stat()	sysconf()	tcdrain()
tcflow()	tcflush()	tcgetattr()	tcgetpgrp()	tcsendbreak()
tcsetattr()	tcsetpgrp()	time()	times()	umask()
uname()	unlink()	utime()	wait()	waitpid()
write()				

– Fonctions POSIX temps-réel

aio_error()	clock_gettime()	sigpause()	timer_getoverrun()
aio_return()	fdatasync()	sigqueue()	timer_gettime()
aio_suspend()	sem_post()	sigset()	timer_settime()

– Fonctions ANSI C

strcpy() strcat() strncpy() strncat()

– Autres fonctions

strlcpy() strlcat() syslog_r()

I.11 Processeurs 64 bits

De plus en plus de processeurs ont une architecture 64 bits. Cela pose de nombreux problèmes aux utilisateurs du langage C. Beaucoup de programmes ne se compilent plus ou pire, se compilent mais ne s'exécutent pas correctement lorsqu'ils sont portés sur une machine pour laquelle les pointeurs sont plus grands qu'un int.

En effet, sur les machines 64 bits, le modèle le plus fréquemment utilisé est celui nommé *LP64* : les types long int et tous les pointeurs sont stockés sur 64 bits, alors que le type int n'utilise « que » 32 bits.

La liste des cas ou l'équivalence entiers/pointeurs est utilisée implicitement est malheureusement trop longue et trop complexe pour être citée entièrement. On ne verra que deux exemples parmi les plus fréquents.

I.11.1 Absence de déclarations des fonctions

De même que pour le type double, à partir du moment où les pointeurs n'ont plus la taille d'un entier, toutes les fonctions passant des pointeurs en paramètre ou retournant des pointeurs doivent être déclarées explicitement (selon la norme ANSI) avant d'être utilisées.

Les programmes qui s'étaient contentés de déclarer les fonctions utilisant des double rencontreront des problèmes sérieux.

I.11.2 Manipulation de pointeurs

Il arrive assez fréquemment que des programmes utilisent le type int ou unsigned int pour stocker des pointeurs avant de les réaffecter à des pointeurs, ou réciproquement qu'ils stockent des entiers dans un type pointeur et cela sans avoir recours à une union.

Dans le cas ou les deux types n'ont pas la même taille, on va perdre une partie de la valeur en le transformant en entier, avec tous les problèmes imaginables lorsqu'il s'agira de lui rendre son statut de pointeur.

I.12 Pré-processeur

Le pré-processeur du langage C (cpp) pose lui aussi certains problèmes et peut être à l'origine de certaines erreurs.

- certaines erreurs de compilation inexplicables proviennent de la re-définition par le pré-processeur d'un symbole de votre programme. N'utilisez jamais d'identificateurs risquant d'être utilisés aussi par un fichier d'en-tête système dans vos programmes. En particulier, tous les identificateurs commençant par le caractère « souligné » (_) sont réservés au système.
- Lors de l'écriture des macros, attention au nombre d'évaluation des paramètres : puisqu'il s'agit de macros et non de fonctions, les paramètres sont évalués autant de fois qu'ils apparaissent dans le corps de la macro, et non une seule fois au moment de l'appel. Ainsi :
  `#define abs(x) ((x)<0?-(x):(x))`
  évalue son argument deux fois. Donc `abs(i++)` incrémentera `i` deux fois.
- Comme dans l'exemple précédent, utilisez toujours des parenthèses autour des paramètres dans l'expansion de la macro. C'est le seul moyen de garantir que les différents opérateurs seront évalués dans l'ordre attendu.

Chapitre II

Un peu d'algorithmique

Le but de cette section est donner quelques pistes pour l'utilisation des algorithmes que vous aurez appris par ailleurs, par exemple dans [6].

II.1 Introduction

Voici en introduction, quelques règles données par R. Pike dans un article sur la programmation en C [7] et reprises récemment dans un livre indispensable [8].

La plupart des programmes sont trop compliqués, c'est-à-dire plus compliqués que nécessaire pour résoudre efficacement le problème qui leur est posé. Pourquoi ? Essentiellement parce qu'ils sont mal conçus, mais ce n'est pas le but de ce document de discuter de conception, le sujet est trop vaste.

Mais l'implémentation des programmes est également trop souvent compliquée inutilement. Là, les quelques règles suivantes peuvent aider à améliorer les choses.

Règle 1 On ne peut prédire où un programme va passer son temps. Les goulets d'étranglement se retrouvent à des endroits surprenants. N'essayez pas d'améliorer le code au hasard sans avoir déterminé exactement où est le goulet.

Règle 2 Mesurez. N'essayez pas d'optimiser un programme sans avoir fait des mesures sérieuses de ses performances. Et refaites-les régulièrement. Si un algorithme plus sophistiqué n'apporte rien, revenez à plus simple.

Règle 3 Les algorithmes sophistiqués sont lents quand n est petit, et en général n est petit. Tant que vous n'êtes pas sûrs que n sera vraiment grand, n'essayez pas d'être intelligents. (Et même si n est grand, appliquez d'abord la règle 2).

Règle 4 Les algorithmes sophistiqués sont plus bugués que les algorithmes simples, parce qu'ils sont plus durs à implémenter. Utilisez des algorithmes et des structures de données simples.

Les structures de données suivantes permettent de traiter tous les problèmes :
– tableaux,
– listes chaînées,
– tables de hachage,
– arbres binaires.
Bien sûr, il peut être nécessaire de les combiner.

Règle 5 Les données dominent. Si vous avez choisi les bonnes structures de données, les algorithmes deviennent presque évidents. Les structures de données sont bien plus fondamentales que les algorithmes qui les utilisent.

Règle 6 Ne réinventez pas la roue.
Il existe des bibliothèques de code disponibles librement sur le réseau Internet pour résoudre la plupart des problèmes algorithmiques classiques. *Utilisez-les !*.
Il est presque toujours plus coûteux de refaire quelque chose qui existe déjà, plutôt que d'aller le récupérer et de l'adapter.

II.2 Allocation dynamique de la mémoire

En plus des pièges cités au paragraphe I.6, on peut observer les règles suivantes :

Évitez les allocations dynamiques dans les traitements critiques. L'échec de malloc() est très difficile à traiter.

Tenez compte du coût d'une allocation : malloc() utilise l'appel système sbrk() pour réclamer de la mémoire virtuelle au système. Un appel système est très long à exécuter.

Allouez dynamiquement les objets dont la taille n'est pas connue d'avance et peut varier beaucoup. Il est toujours désagréable d'imposer une taille maximum à un objet parce que le programmeur a préféré utiliser un tableau de taille fixe.

Libérez au plus tôt les objets non utilisés.

Limitez les copies d'objets alloués dynamiquement. Utilisez les pointeurs.

II.3 Pointeurs

Les pointeurs sont des outils puissants, même si mal utilisés il peuvent faire de gros dégâts, écrivait Rob Pike dans [7] après s'être planté un ciseau à bois dans le pouce...

Les pointeurs permettent des notations simples pour désigner les objets. Considérons les deux expressions :

```
nodep
node[i]
```

La première est un pointeur vers un nœud, la seconde désigne un nœud (le même peut-être) dans un tableau. Les deux formes désignent donc la même chose, mais la première est plus simple. Pour comprendre la seconde, il faut évaluer une expression, alors que la première désigne directement un objet.

Ainsi l'usage des pointeurs permet souvent d'écrire de manière plus simple l'accès aux éléments d'une structure complexe. Cela devient évident si l'on veut accéder à un élément de notre nœud :

```
nodep->type
node[i].type
```

II.4 Listes

Utilisez de préférence les listes simplement chaînées. Elles sont plus faciles à programmer (donc moins de risque d'erreur) et permettent de faire presque tout ce que l'on peut faire avec des listes doublement chaînées.

Le formalisme du langage LISP est un très bon modèle pour l'expression des opérations sur les listes.

Définissez ou utilisez un formalisme générique pour les listes d'une application.

II.5 Ensembles

Les ensembles de taille arbitrairement grande sont un peu difficiles à implémenter de manière efficace. Par contre, lorsqu'on a affaire à des ensembles de taille raisonnable (moins d'une centaine d'éléments) et connue d'avance, il est facile de les implémenter de manière plutôt efficace : l'élément n de l'ensemble est représenté par le bit $n \pmod{32}$ de l'élément $n/32$ d'un tableau d'entiers.

Les opérations élémentaires sur ce type d'ensemble se codent de manière triviale à l'aide des opérateurs binaires `&`, `|`, `~`.

II.6 Tris et recherches

N'essayez pas de programmer un tri. Il existe des algorithmes performants dans la bibliothèque standard C (qsort()). Ou bien utilisez les algorithmes de Knuth [9].

Il en est de même pour les problèmes de recherche de données dans un ensemble. Voici un petit éventail des possibilités :

recherche linéaire : l'algorithme le plus simple. les éléments n'ont pas besoin d'être triés. La complexité d'une recherche est en $O(n)$, si n est le nombre d'éléments dans la liste. L'ajout d'un élément se fait en temps constant. Cela reste la structure adaptée pour tous les cas où le temps de recherche n'est pas le principal critère. Cette méthode est proposée par la fonction lsearch() de la bibliothèque standard C.

arbres binaires : les données sont triées et la recherche se fait par dichotomie. Les opérations de recherche et d'ajout se font en $O(\log(n))$. Il existe de nombreuses variantes de ce type d'algorithmes, en particulier une version prête à l'emploi fait partie de la bibliothèque C standard : bsearch().

tables de h-coding : une fonction de codage (appelée fonction de hachage) associe une clé numérique à chaque élément de l'ensemble des données (ou à un sous-ensemble significativement moins nombreux). Si la fonction de hachage est bien choisie, les ajouts et les recherches se font en temps constant. En pratique, la clé de hachage n'est jamais unique et ne sert qu'à restreindre le domaine de recherche. Une seconde étape faisant appel à une recherche linéaire ou à base d'arbre est nécessaire. Certaines versions de la bibliothèque standard C proposent la fonction hsearch().

II.7 Chaînes de caractères

Les chaînes de caractères donnent lieu à de nombreux traitements et posent pas mal de problèmes algorithmiques. Voici quelques conseils pour une utilisation saine des chaînes de caractères :

- évitez le limiter arbitrairement la longueur des chaînes : prévoyez l'allocation dynamique de la mémoire en fonction de la longueur.
- si vous devez limiter la longueur d'un chaîne, vérifiez qu'il n'y a pas débordement et prévoyez un traitement de l'erreur.
- utilisez de préférence les fonctions de lecture caractère par caractère pour les chaînes. Elle permettent les meilleures reprises en cas d'erreur.
- prévoyez à l'avance l'internationalisation de votre programme : au minimum, considérez que l'ensemble des caractères à traiter est celui du codage ISO Latin-1.
- utilisez les outils `lex` et `yacc` pour les traitements lexicographiques et syntaxiques un peu complexes : vos programmes gagneront en robustesse et en efficacité.

Chapitre III

Créer des programmes sûrs

Bien souvent un programmeur se satisfait d'un programme qui a l'air de fonctionner correctement parce que, apparemment, il donne un résultat correct sur quelques données de test en entrée. Que des données complètement erronées en entrée produisent des comportements anormaux du programme ne choque pas outre-mesure.

Certaines catégories de programmes ne peuvent pas se contenter de ce niveau (peu élevé) de robustesse. Le cas le plus fréquent est celui de programmes offrant des services à un grand nombre d'utilisateurs potentiels, sur le réseau Internet par exemple, auxquels on ne peut pas faire confiance pour soumettre des données sensées en entrée. Cela est particulièrement crucial pour les programmes s'exécutant avec des privilèges particuliers (par exemple exécutés sous l'identité du super-utilisateur sur une machine Unix).

En effet, les bugs causés par les débordements de tableaux ou les autres cas d'écrasement de données involontaires peuvent être utilisés pour faire exécuter à un programme du code autre que celui prévu par le programmeur. Lorsque ce code est le fruit du hasard (des données brusquement interprétées comme du code), l'exécution ne vas pas très loin et se termine généralement par une erreur de type « bus error » ou « segmentation violation ».

Par contre, un programmeur mal intentionné peut utiliser ces défauts en construisant des jeux de données d'entrée qui font que le code exécuté accidentellement ne sera plus réellement le fruit du hasard, mais bel et bien un morceau de programme préparé intentionnellement et destiné en général à nuire au système ainsi attaqué [10].

Par conséquent, les programmes ouverts à l'utilisation par le plus grand nombre doivent être extrêmement vigilants avec toutes les données qu'ils manipulent.

De plus, comme il est toujours plus facile de respecter les règles en les appliquant dès le début, on gagnera toujours à prendre en compte cet aspect sécurité dans tous les programmes, même si initialement ils ne semblent pas promis à une utilisation sensible du point de vue sécurité.

Garfinkel et Spafford ont consacré le chapitre 16 de leur livre sur la sécurité Unix et Internet [11] à l'écriture de programme sûrs. Leurs recommandations sont souvent reprises par

d'autres auteurs.

La revue francophone MISC traite également régulièrement du sujet, par exemple dans [12].

La robustesse supplémentaire acquise par un programme qui respecte les règles énoncées ici sera toujours un bénéfice pour l'application finale, même si les aspects sécurité ne faisaient pas partie du cahier des charges initial. Un programme conçu pour la sécurité est en général aussi plus robuste face aux erreurs communes, dépourvues d'arrières pensées malveillantes.

III.1 Quelques rappels sur la sécurité informatique

Il y a principalement trois types d'attaques contre un système informatique : le vol de mots de passe, les chevaux de Troie et les dénis de service. Les virus informatiques se propagent en général par le mécanisme du cheval de Troie. Cette section se termine par l'analyse d'un exemple simple d'utilisation d'un bug courant dans les programmes pour introduire un cheval de Troie.

III.1.1 Vol de mot de passe

Le vol de mot de passe permet d'utiliser un compte informatique de manière non autorisée sans avoir à mettre en œuvre de processus compliqué pour contourner les mécanismes d'identification du système.

En plus de copier simplement le mot de passe écrit sur un Post-It collé à coté de l'écran ou dans le tiroir du bureau d'un utilisateur, les techniques utilisées pour voler un mot de passe sont :

- l'interception d'une communication où le mot de passe transite en clair,
- le déchiffrement d'un mot de passe codé récupéré soit par interception soit parce qu'il était laissé lisible explicitement,
- l'introduction d'un cheval de Troie dans le code d'un programme chargé de l'identification.

III.1.2 Chevaux de Troie

La technique du cheval de Troie remonte à l'antiquité, mais l'informatique lui a donné une nouvelle jeunesse. Dans ce domaine on appelle cheval de Troie un bout de code inséré dans un programme ou dans des données qui réalise à l'insu de l'utilisateur une tâche autre que celle réalisée par le programme ou les données qui lui servent de support.

Un cheval de Troie peut accomplir plusieurs rôles :

- ouvrir un accès direct au système (un shell interactif). C'était la fonction du cheval de Troie original.
- se reproduire afin de s'installer sur d'autres programmes ou d'autres données afin de pénétrer d'autres systèmes.
- agir sur les données « normales » du programme : les détruire, les modifier, les transmettre à un tiers,...

Le cheval de Troie peut être introduit de deux manières dans le système : il peut être présent dès le départ dans le code du système et être activé à distance, ou bien il peut être transmis de l'extérieur par une connexion a priori autorisée au système.

III.1.3 Déni de service

Ce type d'attaque à tendance à se multiplier. Il peut constituer une simple gêne pour les utilisateurs d'un service, mais peut également servir de menace pour un chantage de plus grande ampleur.

Le déni de service consiste à bloquer complètement un système informatique en exploitant ses bugs à partir d'une connexion a priori autorisée.

On peut bloquer un programme simplement en provoquant son arrêt suite à un bug, si le système n'a pas prévu de mécanisme pour détecter l'arrêt et redémarrer automatiquement le programme en question. D'autre types de blocages sont possibles en provoquant par exemple le passage d'un programme dans un état « cul de sac » dans lequel il continuera à apparaître comme fonctionnel au système de supervision, mais où il ne pourra pas remplir son rôle. Enfin, il peut être possible de provoquer un déni de service en générant avec un petit volume de données en entrée une charge de travail telle sur le système que celui-ci ne pourra plus répondre normalement.

En général le déni de service est une forme dégénérée du cheval de Troie : à défaut de pouvoir détourner un logiciel à son profit, on va se contenter de le bloquer.

III.2 Comment exploiter les bugs d'un programme

Un exposé exhaustif des techniques utilisées par les « pirates » informatiques pour exploiter les bugs ou les erreurs de conception d'un logiciel dépasse largement le cadre de ce document. Nous allons simplement présenter ici un exemple classique de bug qui était présent dans des dizaines (voire des centaines) de programmes avant que l'on ne découvre la manière de l'exploiter pour faire exécuter un code arbitraire au programme qui le contient.

Voici une fonction d'un programme qui recopie la chaîne de caractères qu'elle reçoit en argument dans un buffer local (qui est situé sur la pile d'exécution du programme).

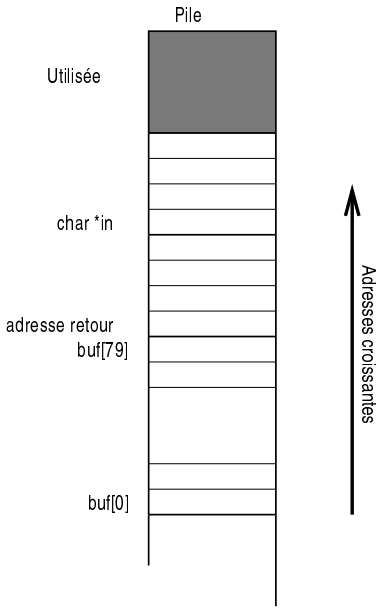
```
int
maFonction(char *in)
{
    char buf[80];

    strcpy(buf, in);
    ...
    return 0;
}
```

Lors de l'exécution de cette fonction, l'organisation des données sur la pile sera celle décrite sur la figure III.2.

Sur cette figure, il apparaît clairement qu'un débordement par le haut du tableau `buf` va écraser sur la pile l'adresse de retour de la fonction. Dans le cas d'une erreur involontaire, cela conduira à un saut à une adresse invalide et provoquera donc une erreur du type *segmentation violation*.

Par contre, on peut exploiter cette lacune pour faire exécuter au programme en question un morceau de code arbitraire. Il suffit pour cela de s'arranger pour que les quatre premiers octets du débordement soient une adresse donnée sur la pile, dans la zone déja allouée, et que le reste

FIG. III.1 – Organisation des données sur la pile lors dans `maFonction`.



du débordement soit un programme en code machine de la bonne longueur pour commencer pile à l'adresse que l'on a mise dans l'adresse de retour.

Ainsi, à la fin de l'exécution de `maFonction`, le processeur va dépiler une mauvaise adresse de retour et continuer son exécution dans le code passé en excès dans le tableau `in`.

Ainsi exposée, cette technique semble assez rudimentaire et difficile à mettre en œuvre. Il est cependant courant de trouver sur Internet dans des forums spécialisés des scripts tout faits capabables d'exploiter ce type de vulnérabilité dans les applications les plus courantes des systèmes existants.

D'autres techniques permettent d'exploiter des débordements de buffers alloués dans le tas (avec malloc()()).

III.3 Règles pour une programmation sûre

La liste des règles qui suivent n'est pas impérative. Des programmes peuvent être sûrs sans respecter ces règles. Elle n'est pas non plus suffisante : d'une part parce qu'il est impossible de faire une liste exhaustive (on découvre chaque semaine de nouvelles manières d'exploiter des programmes apparement innocents), et d'autre part parce que seule une conception rigoureuse permet de protéger un programme contre les erreurs volontaires ou non des utilisateurs.

III.3.1 Éviter les débordements

C'est la règle principale. Il ne faut jamais laisser la possibilité à une fonction d'écrire des données en dehors de la zone mémoire qui lui est destinée. Cela peu paraître trivial, mais c'est

cependant des problèmes de ce type qui sont utilisés dans la grande majorité des problèmes de sécurité connus sur Internet.

Il y a en gros deux techniques pour arriver à cela. Je ne prendrai pas partie pour une technique ou une autre, par contre il est relativement évident que l'on ne peut pas (pour une fois) les mélanger avec succès.

- **Allouer dynamiquement toutes les données.** En n'imposant aucune limite statique à la taille des données, le programme s'adapte à la taille réelle des données et peut toujours les traiter sans risque d'erreur, à condition que la machine dispose de suffisamment de mémoire.
  C'est d'ailleurs là que réside la difficulté majeure de cette méthode. Lorsque la mémoire vient à manquer, il est souvent très délicat de récupérer l'erreur proprement pour émettre un diagnostic clair, libérer la mémoire déjà allouée mais inutilisable et retourner dans un état stable pour continuer à fonctionner.
  Une autre difficulté provient de la difficulté dans certains cas de prédire la taille d'une donnée. On se trouve alors contraint de réallouer plusieurs fois la zone mémoire où elle est stockée, en provoquant autant de recopies de ces données, ce qui ralentit l'exécution du programme.
- **Travailler uniquement avec des données de taille fixe allouées statiquement.** Avec cette technique on s'interdit tout recours à la fonction malloc() ou à ses équivalents. Toutes les données extérieures sont soit tronquées pour contenir dans les buffers de l'application soit traitées séquentiellement par morceaux suffisamment petits. Dans ce cas le traitement des erreurs est plus simple, par contre certains algorithmes deviennent complexes. Par exemple, comment réaliser par exemple un tri de données arbitrairement grandes sans allocation dynamique de mémoire ?

Il faut remarquer ici que les systèmes de mémoire virtuelle aident à gommer les défauts respectifs des deux approches. La possibilité d'allouer des quantités de mémoire bien supérieures à la mémoire physique disponible aide à retarder l'apparition du manque de mémoire dans le premier cas. La faculté de la mémoire virtuelle à ne réclamer de la mémoire physique que pour les données réellement référencées permet dans la seconde approche de prévoir des tailles de données statiques relativement grandes sans monopoliser trop de ressources si les données réelles sont très souvent beaucoup plus petites.

III.3.2 Débordements arithmétiques

Un autre type de débordement qui peut également contribuer à rendre un programme vulnérable (et en tout cas provoquer des erreurs imprévues à l'exécution) est le débordement d'un calcul en arithmétique sur des nombres entiers.

Si le résultat de la multiplication de deux nombres entiers dépasse la valeur maximale qui peut être représentée par le type de données utilisé, le résultat n'est pas défini. En C, sur la plupart des architectures utilisées courament, le résultat sera simplement tronqué de ses bits de plus fort poids si les nombres ne sont pas signés, ou alors, si les nombres sont signés, le débordement vers le bit de signe produira un résultat faux (en général de signe opposé).

Cette erreur peut être exploitée lorsqu'une collection d'objets de même taille doit être allouée par malloc() et que le nombre d'objets de la collection est spécifié par l'utilisateur. Dans ce cas, le produit de la taille d'un objet par le nombre à allouer peut dépasser la valeur maximale représentable par un entier non signé si le nombre d'objets demandé est arbitrairement grand.

L'appel à malloc() qui sera donc fait avec une taille plus faible que celle réellement nécessaire,

peut alors réussir et le programme continuera son exécution en croyant disposer de grandes quantités de mémoire. Le premier accès au dela de la taille réellement allouée provoquera alors un débordement de buffer.

III.3.3 Se méfier des données

Si vous considérez que l'utilisateur de votre programme veut nuire à votre système, toutes les données qu'il fournit en entrée sont potentiellement dangereuses. Votre programme doit donc analyser finement ses entrées pour rejeter intelligemment les données suspectes, sans pénaliser outre mesure les utilisateurs honnêtes.

Le premier point à vérifier a déjà été évoqué : il faut éviter que la taille des données d'entrée ne provoque un débordement interne de la mémoire. Mais il y a d'autres points à vérifier :

- Vérifier les noms des fichiers. En effet l'utilisateur peut essayer d'utiliser les privilèges potentiels de votre programme pour écraser ou effacer des fichiers système.
- Vérifier la syntaxe des commandes. Chaque fois qu'un programme commande l'exécution d'un script, il est possible d'exploiter la syntaxe particulièrement riche du shell Unix pour faire faire à une commande autre chose que ce pourquoi elle est conçue. Par exemple si un programme exécute le code suivant pour renommer un fichier :
  ```
  sprintf(cmd, "mv %s %s.bak", fichier, fichier);
  system(cmd);
  ```
  pour renommer un fichier, si `fichier` contient la valeur :
  ```
  toto toto.bak ; cat /etc/passwd ;
  ```
  la fonction system() va exécuter :
  ```
  mv toto toto.bak ; cat /etc/passwd ;
  toto toto.bak ; cat /etc/passwd;
  ```
  Ce qui va afficher le fichier des mots de passe cryptés du système en plus du résultat initialement attendu.
- Ne pas laisser l'utilisateur fournir une chaîne de format pour les fonctions de type printf(). Utiliser toujours un format fixe ou correctement filtré.
  ```
  printf(buf);
  ```
  est dangeureux si buf est fournit par l'utiliateur. Il faut toujours utiliser
  ```
  printf("%s", buf);
  ```
  à la place.
- Vérifier l'identité de l'utilisateur. Dans tous les cas où cela est possible, l'identification des utilisateurs ne limite pas directement ce qu'il peuvent faire, mais aide à mettre en place des mécanismes de contrôle d'accès.

III.3.4 Traiter toutes les erreurs

Éviter que des erreurs se produisent n'est pas toujours possible.

Prévoyez des traces des problèmes de sécurité, non visibles directement par l'utilisateur. Par contre il est indispensable de prévenir l'utilisateur de l'existence et de la nature de ces traces (loi Informatique et libertés + effet dissuasif)

III.3.5 Limiter les fonctionnalités

Certaines fonctionnalités d'un programme peuvent être dangereuses. Il faut y songer dès la spécification pour éviter de fournir au pirate les moyens de parvenir facilement à ses fins.

– possibilité de créer un shell
– affichage de trop d'information
– traitements sans limites

III.3.6 Se méfier des bibliothèques

Les règles ci-dessus devraient être respectées par les programmeurs qui ont réalisé les bibliothèques de fonctions utilisées par votre application (bibliothèque C standard, interface graphique, calcul matriciel,...). Mais en êtes-vous certains ? L'expérience montre que des problèmes du style de ceux évoqués ici sont présents dans de nombreux logiciels commerciaux, comme dans les logiciels libres.

En général il n'est pas possible d'auditer tout le code des bibliothèques utilisées, soit parce que celui-ci n'est pas disponible, soit simplement par manque de temps et/ou de compétence.

Interrogez-vous sur le type d'algorithme utilisé par les fonctions appelées par votre code. Si l'un d'eux présente des risques de débordement interne, vérifiez deux fois les données, à l'entrée et à la sortie pour détecter du mieux possible les éventuelles tentatives d'attaque. En cas de doute sur un résultat votre programme doit le signaler au plus tôt, et ne pas utiliser ce résultat.

III.3.7 Bannir les fonctions dangereuses

Certaines fonctions de la bibliothèque C standard sont intrinsèquement dangereuses parce que leur sémantique ne permet pas de respecter les règles présentées ci-dessus. Il faut donc s'interdire impérativement de les utiliser. Il peut y avoir des cas où ces fonctions peuvent être utilisées malgré tout de manière sûre. A mon avis, même dans ces cas, il faut les éviter et leur préférer une version sûre. Cela facilite la vérification a posteriori du code, en évitant de provoquer des fausses alarmes qui peuvent être coûteuses à désamorcer. De plus, le raisonnement qui vous a amené à considérer une utilisation d'une fonction dangeureuse comme sûre peut être faux ou incomplet et donc le risque n'est pas éliminé complètement.

Ne pas utiliser	remplacer par	remarque(s)
gets()	fgets()	risque de débordement
scanf()	strtol(), strtod(), strtok(),...	idem
sprintf()	snprintf()	risque de débordement
strcat()	strlcat()	idem
strcpy()	strlcpy()	idem
mktemp()	mkstemp()	section critique : entre la création et l'ouverture du fichier temporaire
system()	fork() & exec()	possibilité d'exploiter le shell

III.4 Pour aller plus loin...

Adam Shostack, consultant en sécurité informatique, a rédigé un guide pour la relecture du code destiné à tourner dans un firewall : http://www.homeport.org/~adam/review.html qui est souvent cité comme référence pour l'évaluation des logiciels de sécurité.

Dan Wheeler a écrit un «Secure programming for Linux and Unix HOWTO» qui fait partie du projet de documentation de Linux : http://www.dwheeler.com/secure-programs/.

Le projet FreeBSD propose un ensemble de recommendations au développeurs qui souhaitent contribuer au projet : http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/secure.html

Matt Bishop a été l'un des précurseurs de la notion de programmation robuste [13]. Ses articles sur le sujet font référence. http://nob.cs.ucdavis.edu/~bishop/secprog.html

Enfin, le chapitre de [11] consacré à la programmation sûre est disponible en ligne : http://www.onlamp.com/pub/a/onlamp/excerpt/PUIS3_chap16/index1.html

Chapitre IV

Questions de style

Les règles présentées ici ne sont pas impératives, il s'agit juste d'exemples de bonnes habitudes qui facilitent la relecture d'un programme.

Ce qui est le plus important, c'est de penser qu'un programme doit pouvoir être lu et compris par quelqu'un d'extérieur, qui ne connaît pas forcément tout du logiciel dont est extrait le morceau qu'il relit. La lisibilité d'un code source (qui peut s'analyser avec les règles de la typographie) est une très bonne mesure de sa qualité.

Les guides de style pour les programmeurs C sont très nombreux dans la littérature. Presque chaque ouvrage sur le langage propose son style. Toutes les grandes entreprises et les grands projets de logiciel ont leurs règles.

Un guide a servi de modèle à de nombreux programmeurs : le *Indian Hill C Style and Coding Standards* des Laboratoires Bell [14]. Ce guide a été amendé et modifié de très nombreuses fois, mais sert de référence implicite commune à de nombreux autres guides.

IV.1 Commentaires et documentation

IV.1.1 Commentaires

Bien commenter un programme est sans doute la chose la plus difficile de toute la chaîne de développement. C'est un des domaines où « le mieux est l'ennemi du bien » s'applique avec le plus d'évidence.

De manière générale, le commentaire permet d'apporter au lecteur d'un programme une information que le programme lui-même ne fournit pas assez clairement.

Pour évaluer la qualité d'un commentaire, le recours aux règles de la typographie est précieux : un commentaire ne doit pas être surchargé de ponctuation ou de décorations. Plus il sera sobre, plus il sera lisible.

Un bon commentaire a surtout un rôle introductif : il présente ce qui suit, l'algorithme utilisé ou les raisons d'un choix de codage qui peut paraître surprenant.

Un commentaire qui paraphrase le code et vient après coup n'apporte rien. De même, il vaut mieux un algorithme bien programmé et bien présenté avec des noms de variables bien choisis pour aider à sa compréhension, plutôt qu'un code brouillon très compact suivi ou précédé de cent lignes d'explications. L'exemple extrême du commentaire inutile est :

```
i++;  /* ajoute 1 a la variable i */
```

D'ailleurs, avec ce genre de commentaires, le risque de voir un jour le code et le commentaire se contredire augmente considérablement.

Enfin, il est sûrement utile de rappeler *quand* écrire les commentaires : tout de suite en écrivant le programme. Prétendre repasser plus tard pour commenter un programme c'est une promesse d'ivrogne qui est très difficile à tenir.

En-têtes de fichiers

Il peut être utile d'avoir en tête de chaque fichier source un commentaire qui attribue le copyright du contenu à son propriétaire, ainsi qu'un cartouche indiquant le numéro de version du fichier, le nom de l'auteur et la date de la dernière mise à jour, avant une description rapide du contenu du fichier.

Exemple (ici l'en-tête est maintenue automatiquement par RCS) :

```
/***
 *** Copyright (c) 1997,1998 CNRS-LAAS
 ***
 *** $Source: /home/matthieu/cvs/doc/cours/C/style.tex,v $
 *** $Revision: 1.11 $
 *** $Date: 2001/03/06 17:50:05 $
 *** $Author: matthieu $
 ***
 *** Fonctions de test sur les types du langage
 ***/
```

Remarque : la notice de copyright rédigée ainsi n'a aucune valeur légale en France. Pour une protection efficace d'un programme, il faut le déposer auprès d'un organisme spécialisé. Néanmoins, en cas de conflit, la présence de cette notice peut constituer un élément de preuve de l'origine du logiciel.

En-têtes de fonctions

Avant chaque fonction, il est très utile d'avoir un commentaire qui rappelle le rôle de la fonction et de ses arguments. Il est également très utile d'indiquer les différentes erreurs qui peuvent être détectées et retournées.

Exemple :

```
/**
 ** insertAfter - insère un bloc dans la liste après un bloc
 **               particulier
 **
 ** paramètres:
 **   list: la liste dans laquelle insérer le bloc. NULL crée une
```

```
**          nouvelle liste
**   pBloc: pointeur sur le bloc a insérer
**
** retourne: la nouvelle liste
**/
```

Donner le type des paramètres ne sert à rien, car la déclaration formelle de la fonction, avec le type exact suit immédiatement.

Commentaires dans le code

On peut presque s'en passer si l'algorithme est bien présenté (voir aussi remarque sur la complexité au chapitre II).

Il vaut mieux privilégier les commentaires qui répondent à la question *pourquoi ?* par rapport à ceux qui répondent à la question *comment ?*.

Les commentaires courts peuvent être placés en fin de ligne. Dès qu'un commentaire est un peu long, il vaut mieux faire un *bloc* avant le code commenté. Pour un bloc, utiliser une présentation sobre du genre :

```
/*
 * Un commentaire bloc
 * Le texte est mis en page de manière simple et claire.
 */
```

Les commentaires placés dans le code doivent être indentés comme le code qu'ils précèdent.

N'essayez pas de créer des cadres compliqués, justifiés à droite ou avec une apparence 3D. Cela n'apporte aucune information, et est très dur à maintenir propre lorsqu'on modifie le commentaire.

IV.1.2 Documentation

Maintenir une documentation à part sur le fonctionnement interne d'un programme est une mission quasiment impossible. C'est une méthode à éviter. Il vaut mille fois mieux intégrer la documentation au programme sous forme de commentaires.

Cette logique peut être poussée un peu plus loin en utilisant dans les commentaires les commandes d'un logiciel de formattage de documents (troff, LaTeX, etc.). Il suffit alors d'avoir un outil qui extrait les commentaires du source et les formatte pour retrouver un document externe avec une présentation plus riche que des commentaires traditionnels. Knuth a formalisé cette approche sous le nom de programmation littéraire [15]

IV.2 Typologie des noms

La typologie des noms (choix des noms des variables, des fonctions, des fichiers) est un élément primordial dans la lisibilité d'un programme. Celle-ci doit respecter plusieurs contraintes :

- **cohérence** choisissez une logique dans le choix des noms de variables et gardez-là.
- **signification** choisissez des noms qui ont un sens en relation avec le rôle de la variable ou de la fonction que vous nommez.
- **modularité** indiquez l'appartenance d'un nom à un module.

– **non-ambiguité** évitez les constructions ambigües pour distinguer deux variables semblables (variations sur la casse par exemple), utilisez des moyens simples (suffixes numériques). Attention, certains éditeurs de liens imposent que les 6 (six !) premiers caractères d'un identificateur soient discriminants.

Il existe plusieurs conventions de choix des noms de variables et de fonctions décrites dans la littérature. Parmi celles, ci on peut citer la « notation hongroise » présentée entre autres par C. Simonyi [16] qui code le type des objets dans leur nom.

Sans entrer dans un mécanisme aussi systématique, il est bon de suivre quelques règles :

– les noms avec un underscore en tête ou en queue sont réservés au système. Ils ne doivent donc pas être utilisés par un utilisateur de base.
– mettre en majuscules les constantes et les noms de macros définies par `#define`. Les macros qui se comportent comme une fonction peuvent avoir un nom en minuscules.
  exemples :
  ```
  #define VITESSE_MAX (1.8)
  #define MAX(i,j) ((i) > (j) ? (i) : (j))
  #define bcopy(src,dst,n) memcpy((dst),(src),(n))
  ```
  `MAX` est identifié comme une macro (et doit le rester). En effet cette macro évalue deux fois l'un de ses arguments. Écrire `max` risquerait de le faire oublier et de conduire à des erreurs.
– mettre en majuscules également les noms de types définis par `typedef` et les noms de structures.
– utiliser de préférence le même nom pour une structure et le type définit pour elle. exemple :
  ```
  typedef struct POS {
      double x;
      double y;
      double theta;
  } POS;
  ```
– les constantes dans les enum commencent par une majuscule.
– les autres noms (variables, fonctions,...) commencent par une minuscule et sont essentiellement en minuscules. Quand un nom comporte plusieurs mots, on peut utiliser une majuscule pour introduire chaque nouveau mot.
– éviter les noms trop proches typographiquement. Par exemple les caractères «l» et «1» sont difficiles à distinguer, il en sera de même des identificateurs «u1» et «ul».
– si une fonction retourne une valeur qui doit être interprétée comme valeur booléenne dans un test, utiliser un nom significatif du test. Par exemple `valeurCorrecte()` plutôt que `testValeur()`.
– la longueur d'un nom n'est pas une vertu en soi. Un index de tableau n'a pas besoin d'être plus complexe que `i`. Les variables locales d'une fonction peuvent souvent avoir des noms très courts.
– les variables globales et les fonctions doivent au contraire avoir des noms qui comportent le maximum d'information. Mais attention, des noms trop longs rendent la lecture difficile.

IV.3 Déclarations

Utilisez le C ANSI, et incluez systématiquement des prototypes des fonctions que vous utilisez. Tous les compilateurs C ANSI ont une option pour produire un avertissement ou une

erreur quand une fonction est appelée sans que son prototype n'ait été déclaré.

Bien entendu, déclarez un type à toutes vos variables et à toutes vos fonctions. La déclaration implicite en entier est une source d'erreurs.

Pour déclarer des types compliqués, utilisez des typedefs. Cela rend le code plus lisible et plus modulaire.

IV.4 Indentation

L'indentation permet de mettre en valeur la structure de l'algorithme. Il est capital de respecter une indentation cohérente avec cette structure. Mais, comme pour la typologie des noms de variables, il n'y a pas de règles uniques.

Personnellement, j'utilise un système d'indentation bien résumé par l'exemple suivant. Il a l'avantage d'une certaine compacité.

```
if (condition) {
    /* 1er cas */
    x = 2;
} else {
    /* 2nd cas */
    x = 3;
}
```

D'autres préfèrent aligner les accolades ouvrantes et fermantes qui se correspondent sur une même colonne :

```
if (condition)
  {
    /* 1er cas */
    x = 2;
  }
else
  {
    /* 2nd cas */
    x = 3;
  }
```

L'incrément de base de l'indentation doit être suffisant pour permettre de distinguer facilement les éléments au même niveau. Quatre caractères semble une bonne valeur.

Il existe plusieurs outils qui maintiennent l'indentation d'un programme automatiquement. L'éditeur `emacs` propose un mode spécifique pour le langage C qui indente les lignes tout seul au fur et a mesure de la frappe, selon des règles programmables.

L'utilitaire Unix `indent` permet de refaire l'indentation de tout un fichier. Un fichier de configuration permet de décrire son style d'indentation favori.

IV.5 Boucles

Évitez absolument de transformer votre code en plat de spaghetti. Le langage C permet de nombreuses constructions qui détournent le cours normal de l'exécution du programme : break, continue, goto...

Toutes ces constructions doivent être évitées dès qu'elles rendent difficile le suivi du déroulement d'un programme. Par la suite, s'il faut prendre un compte un nouveau cas, cela ne pourra se faire qu'en ajoutant des nœuds dans le plat...

Mais attention, dans un certain nombre de cas, notamment le traitement des erreurs, l'utilisation judicieuse d'un break ou d'un goto est plus lisible qu'une imbrication profonde de tests.

IV.6 Expressions complexes

Dé-com-po-sez les expressions trop complexes en utilisant éventuellement des variables intermédiaires. Cela diminue le risque d'erreur lors de la saisie et augmente la lisibilité pour la suite.

Dans la plupart des cas, il est plus efficace de laisser le compilateur optimiser le code et de privilégier la lisibilité du source.

Pour déclarer un type complexe, utilisez plusieurs typedefs intermédiaires.

Par exemple, pour déclarer un tableau de dix pointeurs sur fonctions entières avec un paramètre entier, les deux typedefs suivants sont bien plus lisibles que ce que l'on obtiendrait en essayant de l'écrire directement (laissé en exercice pour le lecteur).

```
typedef int (*INTFUNC)(int);
typedef INTFUNC TABFUNC[10];
```

IV.7 Conversion de types

Attention, terrain glissant. Normalement, il ne devrait pas y en avoir. Avant d'utiliser un cast, demandez-vous toujours s'il n'y a pas un problème dans votre programme qui vous oblige à faire ce cast.

Les pommes ne sont pas des poires, c'est vrai aussi des types informatiques. Si vraiment vous avez des types qui peuvent représenter plusieurs objets différents, les unions sont peut-être un peu plus lourdes à manier, mais elles offrent des possibilités de vérification au compilateur.

En effet, le plus grand piège tendu par les cast, est que vous obligez le compilateur à accepter ce que vous tapez, en lui ôtant tout droit à la critique. Or il est possible de faire des erreurs partout, y compris dans l'utilisation des cast. Mais le compilateur n'a plus aucun moyen de les détecter.

IV.8 Assertions

Le mécanisme des assertions permet de déclarer des prédicats sur les variables d'une fonction qui doivent être vrais (appelés aussi *invariants*). En cas de situation anormale (en général à la suite d'une erreur de logique du programme) l'assertion fausse provoquera un arrêt du programme.

Les assertions sont introduites par le fichier d'en-tête `assert.h` et sont écrites sous la forme :
assert()(*expression*)

Ce mécanisme simple permet d'aider à la mise au point d'algorithmes un peu complexes, à la fois parce qu'ils guident le programmeur pendant le codage et qu'ils permettent d'aider à détecter les erreurs.

Références bibliographiques

[1] S. Summit. *C Programming FAQs : Frequently Asked Questions*. Addison-Wesley, 1995.

[2] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1) :5–48, March 1991.

[3] D.E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[4] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[5] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall, 2nd edition, 1988.

[6] D.E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[7] R. Pike. *Notes on Programming in C*.

[8] B. W. Kernighan and R. Pike. *La programmation en pratique*. Vuibert, 2001.

[9] D.E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[10] Aleph One (`aleph1@underground.org`). Smashing the stack for fun and profit. *Phrack*, N(49), November 1996.

[11] S. Garfinkel, G. Spafford, and Alan Schwartz. *Practical Unix and Internet Security*. O'Reilly and Associates, 3rd edition, 2003.

[12] A. Guignard and P. Malterre. Programmation sécurisée : étude de cas. *MISC 16*, Novembre/Décembre 2004.

[13] M. Bishop. Robust programming. In *ECS153*, 1998.

[14] L.W. Cannon, R.A. Elliot, L.W. Kirchhoff, J.H. Miller, J.M. Milner, R.W. Mitze, E.P. Shan, and N.O. Whittington. *Indian Hill C style and coding standards*. Bell Labs.

[15] D.E. Knuth. Literate programming. *Computer Journal*, 28(2) :97–111, 1984.

[16] C. Simonyi and M. Heller. The hungarian revolution. *Byte*, pages 131–138, Août 1991.

Index