Conseils et erreurs courantes

Samuele Giraudo

Avril 2022

Les points qui suivent ici sont en vrac, sans véritablement de hiérarchie.

1 Vocabulaire

1.1 Comparaisons

"*Inférieur*" veut dire "*inférieur ou égal*". Chose analogue pour "*supérieur*".

1.2 Signe et zéro

"*Positif*" veut dire "*positif ou nul*". Il en est de même pour "*négatif*" qui veut dire "*négatif ou nul*". Ceci est cohérent car le nombre `0` est à la fois positif et négatif. Utiliser "*strictement*" dans les cas précédents pour exclure `0` qui est donc là par défaut.

1.3 Évaluation

Il est impropre de dire qu'une expression "*renvoie*" une valeur. Il faut plutôt dire qu'une expression "*s'évalue*" en une valeur.

1.4 Transmission d'une valeur

Qu'une fonction "*transmette une valeur*" ne veut pas dire qu'elle "*l'affiche sur la sortie standard*". Cela veut dire soit qu'elle la renvoie soit qu'elle l'écrit dans un de ses paramètres (par passage par adresse), au choix.

1.5 Renvoyer une valeur

Une fonction ne "*retourne*" pas une valeur. Une fonction "*renvoie*" une valeur.

1.6 Occurrences

Une *occurrence* d'une valeur `x` dans un tableau `tab` est un indice `i` tel que l'entrée `tab[i]` est égale à `x`. Ainsi, dans le tableau `tab = 3 | 1 | 1 | 4 | 3 | 3 | 1`, les indices `0`, `4` et `5` sont des occurrences de `3`.

2 Fonctions

2.1 Paramètres

Quand il est écrit dans un énoncé qu'il est possible "*d'ajouter d'autres paramètres*" aux fonctions demandées, c'est qu'il faut en général faut le faire.

Par exemple, toute fonction qui prend en paramètre un tableau doit (dans la très grande majorité des cas) prendre aussi sa taille.

2.2 Déclarations de variables locales

Toujours réaliser des déclarations simples, jamais d'initialisation en même temps. On interdit donc les `int x = val;` et analogues. De cette façon, nous avons bien une disjonction stricte entre la partie des déclarations et la partie des instructions d'une fonction.

2.3 Respect des spécifications

Ne pas utiliser de `printf` ou autres choses non demandées explicitement par les spécifications des fonctions. Si une fonction fait plus que ce qui est demandé, elle sera moins utilisable avec d'autres, donc moins générique et moins réutilisable.

2.4 Paramètres constants

Ne pas oublier pas de placer des `const` dans les paramètres là où c'est nécessaire, pour les paramètres qui sont des adresses dont les valeurs pointées ne sont utilisées qu'en lecture.

3 Tableaux

3.1 Tableaux en paramètre

Dans les entêtes de fonctions, pour les paramètres tableaux, ne pas mettre `int tab[]` mais plutôt `int *tab`.

3.2 Test de fin de parcours d'un tableau

Un test `tab[i] == NULL` pour savoir si on a terminé le traitement dans un tableau est bien entendu totalement erroné. Ceci provient d'une confusion entre tableaux et listes. Concernant les tableaux, il faut connaître leur taille pour savoir quand arrêter le traitement.

3.3 Taille

Il est interdit d'effectuer des calculs du type `sizeof(tab) / sizeof(int)` ou autres pour obtenir la taille d'un tableau.

3.4 Tableaux statiques dans les fonctions

Il ne faut jamais renvoyer de tableau statique qui est déclaré comme variable locale d'une fonction. En effet, comme toute variable locale, celle-ci est dans la pile et elle est évacuée en sortant de la fonction.

Il existe une exception à ce propos concernant les variables locales d'un type structurées qui peuvent être renvoyées mais ceci est déconseillé pour des raisons d'efficacité.

3.5 Tableaux statiques de tailles non constantes dans les fonctions

Il n'est pas autorisé en C ANSI de déclarer des tableaux statiques comme variables locales de fonctions qui ont une taille non connue à la compilation. Par exemple,

```
void f() {
    int tab[128];
    ...
}
```

est autorisé car `128` est bien sûr une constante et, lors de la compilation, la taille de `tab` est ainsi connue. En revanche,

```
void g(int k) {
    int tab[k];
    ...
}
```

est interdit car la valeur de `k` est inconnue lors de la compilation (c'est une variable, elle peut par exemple dépendre d'une valeur entrée par l'utilisateur lors de l'exécution) et ainsi la taille de `tab` n'est pas prévisible et donc pas connue à la compilation.

Ces tableaux statiques dont la taille dépend d'une variable sont connus sous le nom de *variable-length arrays* ou *VLA*. Ils n'ont rien de mauvais en soit mais ils ne sont pas en adéquation avec la norme ANSI qui est suivie dans ce module qui demande entre autre que chaque variable possède une taille calculable dès la compilation.

3.6 Création d'un nouveau tableau

Quand il est demandé qu'une fonction "*créé un nouveau tableau*" c'est qu'il faut réaliser une allocation dynamique dans la fonction en question.

3.7 Recherche de dernière occurrence dans un tableau

Quand il faut chercher la dernière occurrence d'un élément dans un tableau, le plus simple est de le parcourir de la droite vers la gauche. C'est dommage de le parcourir dans le sens naturel et de conserver une variable sur l'indice de la dernière apparition.

Pour communiquer le résultat, il faut faire un passage par adresse. Le retour de la fonction renseigne sur la présence ou non de la valeur recherchée. Ainsi, un bon prototype est

```
int dernier_indice_valeur(int *tab, int taille, int x, int *res);
```

où `tab` est le tableau dans lequel la recherche s'effectue, `taille` sa taille, `x` l'élément recherché et `res` un pointeur vers un entier où le résultat sera écrit.

4 Modularisation

4.1 Graphes d'inclusions

Les graphes d'inclusion suivent des conventions bien précises à respecter : on ne met pas les extensions des fichiers car ce ne sont pas des fichiers qui sont mentionnées mais des modules.

Les sommets du graphe sont donc les noms de modules, avec deux types de flèches : les inclusions pleines depuis les `.h` et les inclusions étendues depuis les `.c`.

4.2 Inclusions et transitivité

Les inclusions rendues redondantes par transitivité ne sont pas un problème, bien au contraire.

Par exemple, si `A` inclut `B` et `C`, et `B` inclut `C`, il est faux de dire qu'il n'est pas nécessaire que `A` inclue `C`. En effet, avec une telle configuration, le jour où nous apportons des modifications et que `B` n'a plus besoin d'inclure `C`, la dépendance de `A` à `C` est cassée.

4.3 Dépendances structurelles

Dans les `Makefile` tout `X.o` doit dépendre de `X.c` et de `X.h`, même si la compilation du module s'effectue simplement par `gcc -c X.c`.

5 Gestion des erreurs

5.1 Capture des valeurs problématiques des arguments

Ne pas oublier les pré-assertions et donc les appels à `assert`. Il faut imaginer tous les valeurs problématiques des arguments d'une fonction.

Ceci inclut les tests par rapport à `NULL` des paramètres qui sont des pointeurs notamment quand ils sont voués à être déréférencés dans la suite.

5.2 Position des pré-assertions

Les `assert` ne sont à mettre qu'au tout début du bloc d'instructions d'une fonction (et donc juste après les déclarations de variables locales).

Ce n'est pas totalement faux de mettre des `assert` après, sauf que nous les utilisons dans le cadre de ce cours pour mettre en place des *pré*-assertions. En fait, si le besoin de mettre un `assert` dans le bloc d'instruction se fait ressentir, c'est qu'il faudrait plutôt utiliser un renvoi de code d'erreur.

5.3 Pré-assertions pour restreindre les valeurs des arguments

Sur une spécification d'une fonction dans laquelle on restreint volontairement les valeurs que pourront prendre ses paramètres, il faut souvent vérifier les cas problématiques par des pré-assertions. Par exemple, si on demande une fonction `repeter` paramétrée par une chaîne de caractères non vide `str` et un entier positif `k` et qui répète l'affichage de `str` `k` fois, on écrira

```
int repeter(char *str, int k) {
    /* Declaration des variables. */
```

```
    /* Obligatoire pour eviter les seg. fault a cause des
     * dereferenciations qui vont venir. */
    assert(str != NULL);

    /* On s'assure que str est justement non vide. */
    assert(str[0] != '\0');

    /* Et finalement on s'assure que k est positif. */
    assert(k >= 0);

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

Même si la fonction pourrait marcher dans le cas où `str` serait la chaîne vide, on l'interdit car la spécification le demande.

5.4 Tests des allocations dynamiques

Ne pas oublier pas de tester les retours des `malloc`. Si une allocation s'est mal passée, il faut en tenir compte (renvoi d'une valeur spéciale ou interruption de l'exécution).

5.5 Communiquer une erreur

Dans les fonctions à gestion d'erreur, ne pas quitter l'exécution par un `exit(1)` ou bien par un `exit(EXIT_FAILURE)`. Il faut plutôt renvoyer une valeur spéciale conformément au schéma de gestion d'erreur. Une fonction autre que le `main` ne doit pas avoir la responsabilité de faire stopper l'exécution (sauf rares exceptions).

5.6 Choix du code d'erreur

Quand nous avons une fonction qui renvoie une valeur spéciale en cas d'erreur, il faut s'assurer du fait que cette valeur ne peut jamais être renvoyée en cas de non erreur. Par exemple, si nous avons une fonction de recherche d'occurrence dans un tableau qui renvoie l'occurrence si l'élément est trouvé et `0` sinon serait fausse : `0` pourrait être une bonne réponse. Ici, il faudrait donc renvoyer par exemple une valeur strictement négative.

5.7 Retour du `scanf`

Toujours tester la valeur de retour d'un `scanf`. Si tout s'est bien passé, il doit renvoyer le nombre de lectures correctement réalisées.

6 Opérations bit à bit

6.1 Sur le `xor`

Si nous calculons le `xor` entre deux suites de bits, par exemple

```
a     = 001111000
b     = 010110010
-----------------
a ^ b = 011001010
```

nous pouvons observer que dans le résultat nous avons un bit à `0` si et seulement si les deux bits des deux opérandes sont les mêmes en la position considérée (et, donc, de manière équivalente, nous avons un `1` si et seulement si les deux bits des opérandes sont différents). Ainsi, pour obtenir la distance de Hamming (rappel : il s'agit du nombre de bits en mêmes position mais différents) entre deux suites de bits, il suffit de compter le nombre de bits à `1` du `xor` des deux valeurs. Nous obtenons donc la fonction

```
int hamming(unsigned long long a, unsigned long long b) {
    return nb_1(a ^ b);
}
```

où `nb_1` est une fonction qui renvoie le nombre de bits à `1` de son argument.

6.2 Accès à un bit d'un entier

Un entier de 64 bits (et ceci est valable pour les autres tailles) ne peut pas se gérer comme un tableau. En effet, quand `x` est un entier, il n'est pas possible de lire son bit d'indice `i` par `x[i]` puisque `x` n'est pas un tableau. Il faut procéder comme vu en cours, en utilisant les opérateurs bit à bit `&` et `<<` de la bonne façon.

7 Divers

7.1 Utilisation de `sizeof`

Il est interdit de passer à `sizeof` autre chose qu'un type. L'opérateur `sizeof` ne doit prendre qu'un type comme opérande.

7.2 Déréférenciations et types

L'opérateur de déréférenciation `*` enlève une étoile au type de la variable sur lequel il s'applique.

Par exemple, si `x` est une variable de type `char ***`, alors `*x` est de type `char **` et `**x` est de type `char *`.

Ainsi, quand nous avons une fonction qui doit renvoyer un tableau à deux dimensions d'entiers (donc de type de retour `int **`), si nous déclarons une variable `res` de type `int **` vouée à recueillir le résultat, il faudra au final faire un `return res;` et non pas un `return **res;`.

7.3 Valeur d'une expression

Quand on demande la valeur d'une expression, ce n'est pas la valeur des variables impliquées qui importe, c'est la valeur de l'expression toute entière. Par exemple, dans

```
a = b = 2;
```

nous avons à faire à une instruction dont l'expression sous-jacente a pour valeur `2`. Comme effet secondaire lors de son évaluation (qui est donc la transformation de l'expression toute entière en sa valeur finale), elle affecte `2` à `b` et `2` à `a`.

7.4 Alias pour les types structurés

Il faut utiliser l'alias pour les `typedef struct` si et seulement le type déclaré est récursif (ou mutuellement récursif avec d'autres types).

7.5 Valeurs non signées

Pour forcer une valeur à être positive, ajouter `unsigned`. Ainsi, pour déclarer un type censé représenter des couleurs selon leurs composantes RGB comprises entre `0` et `255`, une bonne solution est

```
typedef struct {
    unsigned char rouge;
    unsigned char vert;
    unsigned char bleu;
} Couleur;
```

7.6 Longueur d'une chaîne de caractères

N'utiliser jamais `strlen` (ou presque). En tout cas certainement pas dans des fonctions qui parcourent des chaînes de caractères. Par exemple, en considérant que `str` est une chaîne de caractères (et donc de type `char *`),

```
for (i = 0; i < strlen(str); ++i) {
    ...
}
```

ce très mauvais code va recalculer la longueur de `str` pour tester la condition de continuation du `for` à chaque tour de boucle. On obtient donc une complexité au minimum quadratique en la longueur de la chaîne. Il faut plutôt exploiter le fait qu'une chaîne de caractères se termine par l'octet `0`. Ainsi, on écrira plutôt

```
for (i = 0; str[i] != '\0'; ++i) {
    ...
}
```

Bien qu'une chaîne de caractère soit un tableau de caractères, comme elle est pourvue d'un marqueur de fin contrairement à ces derniers, il est rarement utile de connaître sa longueur avant de la parcourir.