TRAVAUX PRATIQUES IX

Tu déréférences ou tu pointes ?

Nous revenpns en C. Tous les codes doivent être compilés avec les options `-Wall` et `-Wextra` et `--std=c17` .
On recommande d'ajouter également l'option `-fsanitize=undefined` qui modifie le code produit afin qu'il essaye de détecter des comportements inféfinis (e.g. lire une tableau en dehors de ses bornes).

A Un peu de confort

A.1 Assertions

Afin de simplifier le débugage, on peut vouloir garantir que les préconditions d'une fonction sont vérifiées. Pour cela, on peut utiliser la fonction `assert` au début de la fonction. Elle est rangée dans `assert.h`.

Définition 1 : assert

La fonction `assert` prend en argument une condition (un booléen). Si cette condition est fausse, le programme s'arrête immédiatement (et affiche un message indiquant quelle assertion a échouée). Si la condition est vraie, l'exécution du code continue.

Autrement, dit, un `assert(condition)` revient à `if (!condition) {Fin du programme;}` (notez la négation !).
Par exemple :

C
```c
/** Renvoie la somme de i = 0 à n */
int somme(int n) {
        assert(n >= 0);
        int s = 0;
        for (int i = 1; i <= n; i = i+1) {s = s+i;}
        return s;
}
```

Ici, le `assert` permet de garantir que la fonction ne s'exécute que si `n >= 0`.

On peut bien sûr utiliser des `assert` pour autre chose que des pré-conditions. En fait, les `assert` permettent avant tout de débuguer en vérifiant l'état d'une propriété à un moment donné. Un petit bug en amont pouvant causer un gros bug en aval, il est bon de vérifier en amont que tout est cohérent.

A.2 Structures

Il est possible de créer de nouveaux types en C. La principale façon de le faire est en créant des *structures*. Dans l'idée, un type stricture est un type pour une variable qui « contient des sous-variables ».
Par exemple :

C
```c
struct paire {
        int x;
        int y;
};

struct paire demo;
demo.x = 0;
demo.y = 3;
```

Ici, on a créé un type nommé `struct paire` et une variable `demo` de ce type. Cette variable qui contient deux « sous-variables » : `demo.x` et `demo.y`. On appelle ces sous-variables des *champs*.

Définition 2 : Structures

Un type structure est un type de n-uplets dont chacune des coordonnées sont nommées. On appelle ces coordonnées des champs.
Pour définir un type structure, on utilise :

C
```c
struct nom {
        type0 champ0;
        type1 champ1
        ...
};
```

Si `v` est une variable de ce type, `v.ch` est son champ `ch`. On peut le lire ou le modifier comme une variable normale.

1. Écrire une fonction récursive `struct paire division_euclidienne(int a, int b)` qui renvoie la paire quotient-reste de la division euclidienne de `a` par `b`.
   On utilisera un `assert` pour garantir que `b != 0`.

B La notion de pointeurs

Rappelons les 4 points fondamentaux et distincts qui composent une variable :

— Un identifiant.

— Un type.

— Un contenu, c'est à dire une suite de bits/octets. On parle aussi parfois de valeur.

— Une adresse mémoire, c'est à dire *l'endroit* où est stocké ce contenu.

Jusqu'à présent, en C, nous manipulions surtout les identifiants et un peu les types. Nous n'avons jamais manipulé le contenu directement : c'était C qui traduisait nos valeurs en octets à l'aide du type. Nous n'avons jamais manipulé *tout court* les adresses. En C, une adresse s'appelle un *pointeur*.

Fondamentalement, votre RAM est un immense « tableau » de taille environ 2Go. Une adresse mémoire est un « indice » de ce tableau.

Autre métaphore : votre mémoire est une immense avenue. L'adresse mémoire est le numéro d'une maison (ou d'un terrain) sur cette avenue.

Définition 3 : Pointeur

Un pointeur est une adresse mémoire. `type*` est le type des pointeurs pointant vers des objets mémoire de type `type`. Par exemple, un pointeur pointant vers le contenu d'une variable `int` est de type `int*`.
L'opérateur `&` permet de renvoyer l'adresse mémoire d'une variable. On peut l'utiliser pour définir un pointeur à partir d'une variable :

C
```c
int x = 42;
int* pointeur = &x; // pointeur contient l'adresse de x
```

Réciproquement, l'opérateur `*` permet d'accéder à la valeur du contenu pointée par un pointeur. On parle de déréférencement :

C
```c
int y = *pointeur; // y contient la valeur pointée par le pointeur, donc x
printf("%d\n", y); // affichera donc 42
```

1. **a.** Créer une fonction `int renvoie_valeur_pointee(int* ptr)` qui prend en argument un pointeur vers un entier et renvoie la valeur de cet entier. Testez-la.
   b. Créer une fonction qui prend en argument deux pointeurs vers des entiers et renvoie leur somme. Testez-la.

Nous n'avons pour l'instant qu'effleuré la force de la manipulation des pointeurs. En fait, le déréférencement est beaucoup plus puissant : non seulement il permet de *lire* ce qui est pointé, mais il permet aussi et surtout de le *modifier*.

Proposition 1

L'opérateur de déréférencement `*` permet d'accéder au contenu pointé. On peut donc le modifier :

C
```c
int z = 0;
int* ptr = &z;
*ptr = 666; // modifie ce qui est pointé, c'est à dire le contenu de z
printf("%d\n", z); // affichera donc 666;
```

2. Écrire une fonction `void efface(int* ptr)` qui met à 0 le contenu pointé par `ptr` .

3. Écrire une fonction `void incr(int *ptr)` qui incrémente de 1 l'entier pointé par `ptr` .

Théorème 1

Dans une fonction, si l'on dispose d'un pointeur vers un objet extérieur à la fonction (par exemple un pointeur passé en argument qui pointe sur une variable d'une autre fonction) et que l'on modifie le contenu pointé, on crée un effet secondaire.
De même lorsque l'on modifie une case d'un tableau extérieur à la fonction.

C'est d'ailleurs tout l'intérêt des pointeurs, et toute leur difficulté : travailler par effets secondaires.

4. Dans les deux questions précédentes, quels étaient les effets secondaires ?

On peut garantir l'absence de tels effets secondaires en ajoutant le mot clef `const` au type :

Définition 4

`type const*` est le type des pointeurs pointant vers des objets non-mutable de type `type` .

NB : on croise aussi `const type*`. C'est un style plus ancien, mais qui peut rendre des types moins lisibles plus tard (si type contient lui-même des const).

5. Que fera la fonction suivante (devinez avant de tester !) ?

C
```c
void mouhaha(int const* x) {
        *x = 666;
        return;
}
```

B.1 Retour sur `scanf`

6. Écrire une fonction `void assigne(int x, int* ptr)` qui assigne la valeur `x` à l'entier pointé par `ptr` .

Notre fonction `assigne` est une version simplifiée de `scanf` :

Proposition 2

`scanf` prend en arguments :

- Une chaîne de caractères qui est l'entrée attendue, contenant des spécificateurs `%` (comme `%d` , `%lf` , etc) .
- Des *pointeurs* qui indiquent où seront stockées les valeurs lues par les spécificateurs.

C'est pour cela que l'on mettait des `&` partout dans les `scanf` ! Quand on fait `scanf(" %d", &x)` , on demande de lire un entier (`%d`) et on indique *où* le stocker : à l'adresse `&x` . Retenez l'idée :

- **printf** : pour *afficher*, il faut donner les valeurs qu'il faut afficher.
- **scanf** : pour *lire*, il faut donner *l'adresse* où stocker ce qui est lu.

B.2 Un peu plus dur

7. **a.** Écrire une fonction qui prend en arguments un tableau de deux `double` [1] et echange le contenu des deux cases [2].
   b. Écrire une fonction qui prend en arguments deux pointeurs vers des `double` et échange les contenus pointés. Testez-la.

8. *(bonus)*

1. Rappel : le type `double` est le type des nombres flottants, c'est à dire des « réels ».
2. Pas besoin de pointeurs. Vous savez faire.

a. Que fait la fonction suivante [3] :

C
```c
void wayne(int* ptr_x, int* ptr_y) {
  *ptr_y = *ptr_y + *ptr_x;
  *ptr_x = *ptr_y - *ptr_x;
  *ptr_y = *ptr_y - *ptr_x;
  return;
}
```

b. Vérifiez votre hypothèse en testant.

B.3 Pointeurs et tableaux

Dissipons une confusion usuelle, qui est souvent proférée sur les internets mondiaux ou par des quidams ayant (mal) appris le C :

Proposition 3

Les tableaux ne sont pas des pointeurs, et les pointeurs ne sont pas des tableaux !

Il existe par contre un lien entre les deux. Quand on passe un tableau en argument à une fonction, C fonctionne comme si on passait en fait un pointeur vers le tableau. Ce qui "explique" que le contenu du tableau se comporte comme si passé par référence [4] : modifier le contenu du tableau est un effet secondaire.

On dit qu'un tableau s'affaiblit en pointeur.

B.4 Pointeur `NULL`

De même que l'on initialise souvent un entier à zéro, il est utile d'avoir une valeur pour initialiser « par défaut » un pointeur. Cette valeur, c'est `NULL` (rangée dans `stdlib.h`). Le pointeur `NULL` est le (seul) pointeur qui ne pointe sur rien.

C Chaines de caractères

Définition 5

Une chaîne de caractères est est une suite caractères `char` rangés de manière contigüe en mémoire et se terminant par le caractère nul `\0` . On dit que `\0` est une sentinelle (il marque la fin).
Pour les manipuler, on utilise le type `char*` . C'est à dire qu'une chaine de caractères est représentée par un pointeur vers sa première case. Pour initialiser une chaîne de caractères, on peut donner un texte entre guillemets doubles `"` :

C
```c
char* texte = "Un seul être vous manque, et tout est dépeuplé !\n"
```

'U'	'n'	' '	's'	...	'!'	'\n'	'\0'

Pour lire la case d'indice `i` d'une chaîne de caractères, la syntaxe est la même que pour un tableau : `texte[i]` . On ne peut pas la modifier.
On appelle longueur d'une chaîne de caractère le nombre de caractères qu'elle contient, de son premier caractère inclus jusqu'à `\0` exclu.
Pour encoder un texte quelconque, on l'encode en Unicode puis on interprète cet unicode comme une chaîne de caractères `char` (quitte à utiliser plusieurs `char` pour coder un Unicode de plus de 1 octet).
Pour afficher une chaine de caractères, on utilise le spécificateur `%s` .

Par contre, on peut *penser* une chaîne de caractère comme un tableau (non-mutable) de longueur inconnue : ce sont des cases contigues (comme un tableau !) qu'il faut lire les unes après les autres jusqu'à atteindre `\0` .

9. Écrire une fonction `int chaine_longueur(char const* texte)` qui prend en argument une chaine de caractères et renvoie sa longueur.

3. Le nom n'est pas une référence (haha) au sosie de Batman. Si vous avez la véritable référence, vous avez du goût. Sinon... vous avez probablement du goût aussi.
4. Et on ne détaillera pas plus !

Subtilités

Les chaines de caractères sont un nid à fausses intuitions.

Longueurs contre-intuitives La longueur d'une chaîne de caractères n'est pas évidente. Par exemple, `"a"` a une longueur 1 mais `"ä"` a une longueur 2 .

10. Pourquoi ?

C'est pour ce type de considérations que l'on n'utilise pas des tableaux de longueur connue pour stocker du texte : un humain peut difficilement prévoir la longueur d'une chaîne de caractères.

Confusion avec les tableaux Le code `char* s = "hello";` crée bien une chaine de caractère. Et bien que l'on puisse accéder à chaque élément avec la syntaxe `[]` , ce n'est PAS un tableau.

Ainsi, `char* sbis = {'h', 'e', 'l', 'l', 'o', \0};` est faux et renverra une erreur. Par contre, `char[6] sbis = {'h', 'e', 'l', 'l', 'o', \0};` est correct mais ne crée pas une chaine de caractères (et on ne pourrait donc pas [5] le passer en argument à notre fonction `longueur`).

Non-mutabilité Une chaine de caractère définie par `char* s = "...";` n'est pas mutable. Ainsi :

C
```c
char* s = "bonjour";
s[0] = 'B'; // J'avais oublié la majuscule...
```

renverra une erreur à la deuxième ligne. Corollaire : le contenu d'une chaine de caractères doit lui être donnée à son initialisation. **On devrait donc penser les chaines de caractères comme des `char const*` .**

Théorème 2

> Pour ne pas vous emmeler les pinceaux, déclarez toutes vos chaines de caractères comme des `char const*` .

Mais parfois ça marche quand même Avec les outils de la semaine prochaine, nous pourrons créer des vraies-fausses chaines de caractères mutables, de longueur connue. Le plus confusant sera que leur type sera... `char*` aussi [6]. Mais si vous prenez l'habitude de `char const*` vous ne ferez pas d'erreurs [7] [8].

Un peu de tri

11. Écrire une fonction `bool inf_lexico(char const* mot_gauche, char const* mot_droite)` qui prend en argument deux chaines de caractères [9] ; et renvoie `true` si la première est lexicographiquement inférieure à la seconde (c'est à dire si le premier mot serait avant le second dans le dictionnaire), et renvoie `false` sinon.

5. Bon, en fait si, avec beaucoup de bonne volonté on peut. Mais faites comme si on ne pouvait pas. Laissez moi vous simplifier la vie, d'accord ?
6. La vie est parfois mal faite.
7. Honnêtement, les chaînes de caractères sont un nid à erreurs dans quasiment tous les langages existants. Cela provient du fait que l'encodage unicode a une longueur variable... allez faire un joli type pour le texte avec cela.
8. Par contre la longueur variable de l'encodage Unicode est là pour une très pertinente raison ! On en reparlera quand on parlera d'encodage prefix-free avec Huffman.
9. Pour vous aider, imaginez que ce sont des caractères ASCII et qu'il n'y a pas d'espaces. Bref, des mots gentils.