Programmation en C
TP n° 1 : Introduction

Augustin Laouar

I) Remerciements

Les exercices présentés dans ce TP sont tirés des travaux pratiques réalisés par François Pitois. Je le remercie chaleureusement de m'avoir autorisé à les partager.

II) Exercices d'introduction au C

Exercice 1 : Comprendre le C.

Voici un exemple de programme écrit C :

```
1  #include <stdio.h>
2
3  int somme(int tableau[], int taille){
4      int i;
5      int accumulateur;
6      accumulateur = 0;
7      for (i = 0; i < taille; i++) {
8          accumulateur = accumulateur + tableau[i];
9      }
10     return accumulateur;
11 }
12
13 int main(){
14     int notes[5] = {12,18,15,13,20};
15     int total = somme(notes, 5);
16     printf("Le␣total␣est␣:␣%d.␣Au␣revoir.\n", total);
17     return 0;
18 }
```

1. Recopiez ce programme dans un éditeur, compilez ce programme puis exécutez-le.
2. Supprimez la ligne 1 et essayez de compiler. Ça ne fonctionne pas. Déduisez-en à quoi sert cette ligne.
3. D'après cet exemple, comment fait-on pour déclarer une fonction en C ? En particulier, comment précise-t-on le type de retour de la fonction ?
4. À quoi servent les lignes 4 et 5 ?
5. Modifiez la ligne 15 en remplaçant le nombre 5 par 4. Essayez ensuite les valeurs 6 et 123456. Avez-vous une idée de ce qui se passe ?
6. Que fait la ligne 16 ?
7. En vous inspirant de la fonction `somme`, écrivez une fonction `affiche` qui affiche le contenu d'un tableau. En C, pour écrire une fonction qui ne renvoie rien, on utilise le type `void`. La déclaration de la fonction sera donc `void affiche(int tableau[], int taille)`.

Exercice 2 : Calculatrice.

On veut écrire un programme `calc_sum` qui, lorsque l'utilisateur rentre une ligne contenant des entiers (en représentation décimale) séparés par des symboles `+`, comme `"7+52+3\n"` ou `"2+6+74+13\n"`, le programme affiche le résultat de la somme. Si par contre la ligne rentrée n'a pas le bon format, `calc_sum` doit se terminer en affichant `"Bye."`. Par exemple, si on rentre la chaine `"3+8\n"` le programme affiche `"11"` et attend une nouvelle ligne, mais si on rentre la chaine `"+7+8\n"` ou `"4+7+8+\n"` le programme termine en affichant `"Bye."`.

1. Écrivez le programme `calc_sum`.
2. Modifiez le programme `calc_sum` afin qu'il accepte les nombres décimaux.

Exercice 3 : Types et transtypage.

Il existe plusieurs types en C. Vous connaissez le type `int` qui correspond à un entier, `float` à un nombre à virgule flottante, `char` à un caractère. Il est possible de passer d'un type à l'autre. Cela s'appelle un *transtypage*.

1. Écrivez la fonction suivante : `float div(int a, int b){return a / b;}` Obtient-on toujours le résultat attendu ?

Pour obtenir le résultat souhaité, il faut que l'un des deux opérandes de la division soit un flottant.

2. Modifiez la fonction en `float div(int a, int b){return (float) a / b;}` et testez cette nouvelle fonction.

Le résultat ainsi obtenu est correct. L'instruction `(float)` permet de convertir l'entier a en le nombre flottant a. Il s'agit d'un transtypage (*cast* en anglais). La syntaxe consiste à mettre le nom du type de destination entre parenthèses. Certains transtypages sont implicites, d'autres doivent être explicités sinon ils produisent un *warning* à la compilation.

3. Écrivez les instructions `float x = 7;` `int n = 3.5;` `int* px = &x;` et `float* pn = &n;` Quels transtypages sont implicites ? Lesquelles émettent un *warning* ?

On peut enlever le *warning* en explicitant le transtypage.

4. Modifiez les instructions qui émettaient un *warning* en ajoutant un transtypage explicite.

Un type peut être modifié en ajoutant le qualificateur `const`. Cela permet d'empêcher la variable d'être modifiée.

5. Écrivez l'instruction `const int n = 7;` puis essayez de modifier n.
6. Écrivez l'instruction `const int* p = &n;` puis essayez de modifier la valeur pointée par p.
7. Écrivez l'instruction `int* p = &n;` Pourquoi y a-t-il un *warning* ?

En C, les types `void` et `void*` sont assez particuliers. Le type `void` correspond au type vide. C'est ce que l'on utilise quand une fonction ne renvoie rien. On peut aussi l'utiliser quand une fonction ne prend aucun argument. Vous trouverez par exemple la syntaxe `int main(void)` pour indiquer que la fonction `main` ne prend pas d'argument. C'est équivalent à `int main()`. Le type `void*` permet d'indiquer que l'on a un pointeur, mais que l'on ne connait pas le type de l'objet pointé. C'est ce qui permet d'avoir un semblant de fonctions polymorphique en C. Nous allons regarder deux exemples dans les exercices qui suivent.

Exercice 4 : Allouer de la mémoire.

La librairie `stdlib.h` contient les fonctions permettant d'allouer et de libérer de la mémoire sur le tas. Nous commençons par nous intéressez à la fonction `malloc` qui permet d'allouer de la mémoire. Sa signature est la suivante : `void* malloc(size_t memorySize);` Le type `size_t` est un type qui peut contenir un entier positif. Comme pour les autres types qui stockent des entiers, le transtypage se fait implicitement vers le type `int`. La fonction `malloc` renvoie un `void*` car il ne connait pas le type pointé.

1. Écrivez l'instruction `float* p = malloc(sizeof(float));` À votre avis, que fait la fonction `sizeof` ?

Quand on alloue de la mémoire via `malloc`, il faut la libérer quand on ne l'utilise plus. On utilise pour cela `free`, qui a pour signature : `void free(void* pointer);`

2. Libérez la mémoire que vous avez allouée à la question précédente.

L'intérêt est en général d'allouer de la mémoire pour un tableau.

3. Allouez la mémoire pour un tableau de taille 100, modifiez l'élément en 30^{e} position, puis libérez la mémoire. Essayez d'accéder ensuite à l'élément en 30^{e} position.

III) Exercices intermédiaires

Exercice 5 : Trier un tableau.

En C, il existe une implémentation polymorphique du tri rapide. Elle a la signature suivante :

```
void qsort(void* array, size_t elementCount, size_t elementSize,
           int (*compareFunction)(const void*, const void*));
```

Le paramètre `array` est un pointeur vers la première case du tableau, `elementCount` est le nombre d'éléments dans le tableau et `elementSize` est la taille d'un élément. Quant à `compareFunction`, il s'agit d'une fonction qui prend en argument deux pointeurs et qui renvoie quel élément pointé est le plus grand. Elle doit renvoyer un nombre strictement positif si c'est le premier, un nombre strictement négatif si c'est le second, et 0 s'ils sont égaux. Note : Comme il est interdit de déréférencer un pointeur `void*`, il faut systématiquement transtyper ce pointeur vers un pointeur (constant) du type voulu.

1. Implémentez une fonction de comparaison d'entiers que l'on pourra utiliser avec `qsort`.
2. Utilisez `qsort` pour trier un tableau d'entiers.

Exercice 6 : Le crible d'Ératosthène.

On rappelle qu'un entier naturel p est dit *premier* s'il n'existe aucun entier naturel autre que 1 et p lui-même qui le divise. Un nombre qui n'est pas premier est *composé*. Par convention, 1 n'est ni premier ni composé. Dans cet exercice, nous écrirons une fonction calculant la liste de tous les nombres premiers plus petits qu'un certain nombre n donné en entrée. Pour ce faire, on utilisera une méthode connue sous le nom de *crible d'Ératosthène*, qui consiste à déterminer les nombres premiers par filtrages successifs. L'idée est de repérer les nombres composés et de déduire les nombres premiers par complémentarité : pour chaque nombre k supérieur ou égal à 2, on marque chaque multiple de k autre que k lui-même comme étant *composé*. En faisant ça jusqu'à une certaine borne, on déduit tous les nombres premiers inférieur ou égaux à n.

1. Écrivez un programme `eratosthene` qui prend en entrée un entier `n` et affiche tous les nombres premiers de 2 à n calculés en appliquant le crible d'Ératosthène. N'hésitez pas à factoriser votre programme en plusieurs fonctions auxiliaires.
2. Modifiez votre programme pour qu'il affiche aussi la fonction $\pi(n)$ de compte des nombres premiers, c.à-d. le nombre de nombres premiers strictement inférieurs à `n`. [1]

Exercice 7 : Labyrinthes

On veut créer un algorithme aléatoire qui puisse générer des labyrinthe tels que celui ci-dessous :

```
XXXXXXXXXXXXXXX
X X     X     X
X XXX X X X XXX
X X   X X X   X
X X XXXXX XXX X
X X   X     X X
X X X X XXXXX X
X X X X X   X X
X XXX X X X X X
X X   X X X   X
X X XXX X XXX X
X X     X X X X
X XXXXXXX X X X
X         X   X
XXXXXXXXXXXXXXX
```

1. Estimer de façon aussi précise que possible ce nombre $\pi(n)$ est un des problèmes centraux de la branche des mathématiques appelée *théorie des nombres*. Par exemple, si vous arrivez à démontrer que $\left|\pi(n) - \int_0^n \frac{dt}{\log t}\right| < \sqrt{n}\log n$ pour tout $n \geqslant 2$, vous aurez atteint la gloire éternelle (ainsi que gagné 1 million de dollars). Du point de vue numérique, nous avons vu comment calculer "rapidement" $\pi(n)$ pour n de l'ordre de millions (essayez `eratosthene(1000000)` et regardez combien de temps cela prend). Pour aller au delà, il faut des méthodes plus sophistiquées.

Pour ce faire, on va produire un tableau 2D nommé A, de taille $n \times m$, où n et m sont des entiers impairs. On impose les contraintes suivantes :

- chaque cellule $A[i, j]$ vaut `'X'` ou `'␣'`,
- pour tout i, j, les cellules $A[0, j], A[i, 0], A[n-1, j]$ et $A[i, m-1]$ contiennent `'X'`,
- si i et j sont tous les deux pairs, on a $A[i, j] =$ `'X'`,
- si i et j sont tous les deux impairs, on a $A[i, j] =$ `'␣'`,
- le graphe G est un arbre, où G est le graphe non orienté (V, E), avec V qui est l'ensemble des couples (i, j) dans $\{1, 3, \ldots, n-2\} \times \{1, 3, \ldots, m-2\}$ (c'est-à-dire tels que i et j sont tous les deux impairs) et E qui contient les arêtes $\{(i, j), (i', j')\}$ telles que :
  — $i' = i + 2$ et $j' = j$ et $A[i + 1, j] =$ `'␣'`,
  — ou $i' = i$ et $j' = j + 2$ et $A[i, j + 1] =$ `'␣'`.

1. Vérifiez que l'exemple donné ci dessus respecte bien les contraintes énoncées.

2. Écrivez un programme qui permet de générer de tels labyrinthes.

IV) Exercies avancés

Exercice 8 : Un peu de Caml en C.

On veut implémenter certaines fonctionnalités de Caml en C. Une des fonctions phares de Caml est la fonction `map` de signature `('a → 'b) → 'a list → 'b list` qui applique une fonction à l'ensemble des éléments d'une liste. On se propose d'implémenter une variante de cette fonction, qui applique une liste de fonctions à un même élément. Sa signature en Caml serait `('a → 'b) list → 'a → 'b list`. On va implémenter cette fonction dans le cas `'a = 'b = int`.

1. Écrivez une fonction qui prend en argument une fonction f : `int → int` ainsi qu'un entier n et qui renvoie $f(n)$. En C, pour déclarer un objet f de type `int → int`, on déclare `"int␣(*f)(int);"`.
2. Définissez une structure de liste chainée. On aura à la fois besoin de listes chainées d'entiers et de liste chainées de fonctions de type `int → int`. Faites en sorte que la même structure puisse être utilisée dans les deux cas en utilisant `void*`.
3. Écrivez les fonctions classiques sur les listes chainées, à savoir la création d'une liste vide, l'ajout d'un élément en tête, le test qui renvoie si une liste est vide, la fonction qui récupère l'élément en tête de liste et celle qui renvoie la queue de la liste.
4. Écrivez une fonction *récursive* qui prend en paramètre une liste chainée L de fonctions $L[i]$: `int → int` ainsi qu'un entier n et qui renvoie la liste chainée des applications de chaque fonction $L[i]$ à n.

Exercice 9 : Langage BrainFuck.

Le BrainFuck est un langage de programmation exotique très proche d'une *machine de Turing*. Dans ce langage, on imagine que l'on a une mémoire infinie, organisée sous forme d'un tableau bi-infini, où chaque case est de la taille d'un octet. On a un unique pointeur qui pointe vers une case de la mémoire. Les quelques instructions qui existent permettent de déplacer ce pointeur, de lire la valeur pointée, de la modifier, de gérer les entrées/sorties et de faire des boucles d'exécution.

`>`	Incrémente de 1 le pointeur.
`<`	Décrémente de 1 le pointeur.
`+`	Incrémente de 1 l'octet pointé.
`-`	Décrémente de 1 l'octet pointé.
`.`	Affiche dans la console l'octet pointé sous forme ASCII.
`,`	Lit l'entrée standard et stocke l'octet lu à la position pointée.
`[`	Si l'octet pointé est égale à 0, saute à l'instruction après le `]` correspondant.
`]`	Retourne au `[` correspondant.

FIGURE 1 – Liste des instructions du BrainFuck.

```
++++++++++[>+++++++>++++++++++>+++>+<<<<-]
>++.>+.+++++++..+++.>++.<<+++++++++++++++.>.+++.---.----.>+.>.
```

FIGURE 2 – Exemple de programme BrainFuck. Le retour à la ligne ne fait pas partie du code.

1. Programmez un *traducteur* BrainFuck vers C. Votre programme doit lire un fichier `in.bf` contenant du BrainFuck et doit créer un ficher `out.c` contenant du C. Si on compile `out.c` et qu'on exécute le binaire produit, on doit obtienir la même chose que si on avait exécuté `in.bf`.

Conseil : d'abord, en supposant que l'on ait un tableau de taille bi-infini et une variable `pointeur`, trouvez comment on pourrait traduite chacune des 8 instructions du BrainFuck en du code C. Ensuite, gérez le fait que le tableau soit fini en le redimensionnant dès que `pointeur` sort du tableau.

2. Programmez un *interpréteur* BrainFuck. Votre programme doit lire un fichier `in.bf` et exécuter son code instruction par instruction.
3. On veut visualiser ce qui se passe quand le code est interprété. Modifiez votre programme pour afficher à chaque instant le contenu du tableau bi-infini qui se trouve autour de l'octet pointé ainsi que l'endroit où l'on est dans le code.