Structures de données
TD1 Rappels de C

1 Exercice 1

Dans ce cours, vous veillerez à respecter au mieux les conventions d'écriture du langage C ANSI (KR) décrites sur le poly du TD ainsi qu'à bien nommer vos variables et fonctions. Ces deux consignes sont à la fois pour aider la compréhension des éventuels relecteurs mais aussi pour vous, pour déboguer plus facilement ou lorsque vous reprenez votre code après plusieurs jours/semaines/mois...

N'oubliez pas des rester constant sur l'indentation : soit vous utilisez des tabulations, soit vous utilisez toujours le même nombre d'espaces.

2 Exercice 2

2.1 Compilation et exécution

```
gcc fichier.c -o programme
./programme
```

2.2 Nombre aléatoire

La fonction **int rand() ;** (dans stdlib.h) retourne un nombre aléatoire entre **0** et la constante **RAND_MAX**.

Si vous exécutez plusieurs fois votre programme, vous obtiendrez toujours les mêmes nombres aléatoires. Cela est du au fait que nous utilisons un générateur de nombres dit pseudo aléatoires. Nous pouvons remédier à ce problème en initialisant l'algorithme du générateur avec une graine (= un état) aléatoire. Pour cela nous utilisons les fonctions :

- **unsigned time(time_t* t) ;** (dans time.h) retourne la date courante en secondes (donc différente à chaque appel) si le pointeur t est NULL. Nous utiliserons cette valeur comme graine aléatoire.
- **void srand(unsigned graine) ;** (dans stdlib.h) pour initialiser le générateur. Cette fonction n'est à appeler qu'une seule fois, avant le premier rand().

2.2.1 Nombre entier aléatoire

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
int main(int argc, char* argv[])
{
    // initialisation du generateur aleatoire a la date courante
    srand(time(NULL));
    // affichage d'un nombre aleatoire (different a chaque execution)
    int nombre = rand();
    printf("nombre : %d\n", nombre);
    return 0;
}
```

2.2.2 Nombre entier borné

```c
int min = 5;
int max = 10;

int nombre1 = rand() % (max-min) + min; // max exclu, je ne peux pas avoir 10
int nombre2 = rand() % (max-min+1) + min; // max inclus, je peux avoir 10
```

2.2.3 Nombre réel entre 0 et 1

```c
float nombre0 = rand() / RAND_MAX; // FAUX !!!
float nombre1 = rand()*1.0 / RAND_MAX;
float nombre2 = (float)rand() / RAND_MAX;
float nombre3 = rand() / (float)RAND_MAX;
float nombre4 = rand() / (RAND_MAX + 0.0);
```

Lorsque les deux termes de votre division sont des entiers, vous faites la division entière, le résultat est donc un entier. Pour remédier à cela, un des termes doit être un nombre flottant, soit en le multipliant par **1.0** (implicit cast) soit en forçant sa conversion (explicit cast).

2.2.4 Nombre réel borné

```c
int min = 5;
int max = 10;
float nombre1 = ((float)rand() / RAND_MAX) * (max-min) + min; // max exclu
float nombre2 = ((float)rand() / RAND_MAX) * (max-min+1) + min; // max inclus
```

2.3 scanf

La fonction **int scanf(char* format, ...) ;** permet de récupérer une ou plusieurs données formatées saisies par l'utilisateur.

```
int min, max;
printf("entrez le min et le max, separes par un espace : ");
scanf("%d %d", &min, &max); // printf avant pour afficher un message
printf("min = %d max = %d\n", min, max);
```

3 Exercice 3 Complexité

- **compare_tab** est en $O(n)$ car dans le pire cas (les tableaux sont identiques) la boucle fait n itérations (et chaque itération est en temps constant).
- **produit_ou_somme** est en $O(1)$ (=temps constant) car le nombre d'opérations est constant, il ne dépend pas des données.
- **compare_mat** dans le pire cas (les matrices sont identiques) la boucle while fait n itérations et chaque itération fait appel à 2 opérations élémentaires (en $O(1)$) et à compare_tab(...) dont la complexité est en $O(n)$. La complexité d'une itération est donc en $O(2+n) = O(n)$. Au final, la complexité de la fonction est en $O(n * n) = O(n^2)$.

Si vous avez une fonction composée de deux boucles (while, for, do while, peu importe) non imbriquées, parcourant chacune n itérations en $O(1)$ dans le pire des cas alors cette fonction est en $O(n)$.

```
void test1(int n) {
    for(int i = 0; i < n; i++) {
        printf("%d\n", i);
    }
    for(int j = 0; j < n; j++) {
        printf("%d\n", j);
    }
    // Complexite en O(n)
    // meme si je rajoute une troisieme boucle ci-dessous
}
```

En revanche, si ces deux boucles sont imbriquées (typiquement lorsque vous parcourez un tableau en 2d), alors votre fonction est en $O(n^2)$. Si vous avez 3 boucles imbriquées parcourant chacune n itérations alors la fonction est en $O(n^3)$ ...

```
void init_tab_2d(int n, int** matrice) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            matrice[i][j] = i * j;
        }
    }
    // Complexite en O(n^2)
}
```

4 Exercice 4 Pointeurs et fonctions

4.1 Allocation mémoire

```
int* p = (int*) malloc(sizeof(int));
```

La variable p est de type *pointeur de int*. La fonction malloc() alloue dans la mémoire une zone de 4 octets (sizeof(int) = 4) et retourne l'adresse mémoire de cette zone qui est affecté à p (la valeur d'un pointeur est donc une adresse).

```
printf("%d\n", *p);
```

Pour lire ou écrire dans la zone mémoire pointée par p, il faut déférencer le pointeur pour cela on utilise la notation $*p$. Dans le cas du printf ci-dessus, comme nous n'avons pas initialisé le nombre entier celui-ci peut avoir n'importe quelle valeur.

4.2 Passage par valeur vs passage par référence

```
int incrementer(int i) {
    return i++;
}
int main() {
    int nb = 7;
    printf("%d\n", incrementer(nb));
    // affiche 8 mais nb vaut toujours 7
    return 0;
}
```

Quand vous appelez une fonction comme celle ci-dessus, ces arguments sont recopiés dans la mémoire, c'est ce qu'on appelle le passage par valeur. Si

dans la fonction je modifie la valeur des arguments cela n'aura pas d'impact sur les variables de la fonction appelante (ici main()).

Une autre façon de faire est le passage par référence qui consiste à passer en arguments des pointeurs. De cette façon, la modification de la valeur d'un argument a un impact sur le reste de mon programme (voir ci-dessous).

```
void incrementer(int* i) { // mon argument est un pointeur
    *i++; // je dereference et j'incremente
}
int main() {
    int nb = 7;
    incrementer(&nb); // & permet de passer la reference de nb (= l'adresse)
    // nb vaut maintenant 8
    printf("%d\n", nb);
    return 0;
}
```

Pourquoi faire ça ?

- Permet de retourner plusieurs valeurs
- Evite de recopier (voir de saturer) la mémoire lorsque vous souhaitez utiliser une grande zonne mémoire dans une fonction (typiquement une base de données, un fichier, une grosse structure...)

5 Exercice 5 Pointeurs et Tableaux

```
int tab1[5]; // tableau statique de 5 entiers
int* tab2 = (int*) malloc(5 * sizeof(int)); // tableau dynamique de 5 entiers

// toujours liberer la memoire quand on ne l'utilise plus (1 malloc = 1 free)
free(tab2);
```

Il existe deux façons pour accèder aux éléments d'un tableau dynamique. Par exemple pour accèder à la troisième case (donc indice 2) :

- $tab[2]$ façon tableau statique
- $*(tab + 2)$ façon arithmétique des pointeurs **déconseillé sauf cas exceptionnels ! !**

Contrairement à d'autres langages (python, java...), il n'est pas possible d'obtenir la taille d'un tableau après coup, c'est à vous de conserver cette information. Il y a plusieurs façons de faire, la meilleure étant d'encapsuler la taille et le tableau dans une structure.

```
typedef struct {
    int* tab;
    int taille;
} intTab;
```