TD : Programmation en C — semaine 3

23 février 2018

Objectif(s)

- ✯ Structure de données : les tableaux.
- ✯ Déclaration et appel de fonctions.

Exercice 1 – Remise en forme

Note : Pour tous vos programmes, vous utiliserez le script *run.sh* pour compiler.

1. Afficher un triangle rectangle avec des étoiles ; vous demanderez à l'utilisateur la hauteur. Par exemple avec une hauteur de 4 :

   ```
   *
   **
   ***
   ****
   ```

2. Même question avec un échiquier de taille 8×8. Pour afficher des cases blanches et noirs, vous pouvez utiliser le caractère Unicode correspondant, dans notre cas `\u25A1` pour une case blanche et `\u25A0` pour une noire.

Exercice 2 – En ligne !

Une variable est définie par un couple $(addr, size)$ tel que $addr$ est son adresse en mémoire et $size$ la taille de la case mémoire à cet endroit. Vous pouvez imaginez qu'un programme est une fonction qui va manipuler la mémoire avec trois opérations principales :

— Allocation dans la mémoire. Exemple : `int v;` va créer un couple $(addr, size)$ tel que $addr$ sera l'adresse de v et $size$ aura une taille de `sizeof(int)` où `sizeof` est une fonction qui donne la taille d'un type.
— Lire dans la mémoire. Exemple : `x = v;` va lire la valeur de v à l'adresse `&v` (pour ensuite faire une écriture sur x).
— Écrire dans la mémoire. Exemple : `v = 1;` va écrire la valeur 1 dans la case à l'adresse `&v`.

1. Expérimentez ces différentes opérations en affichant l'adresse, la valeur et la taille de plusieurs variables avec des types différents (`char`, `int`, `float`, `double`,...). Les spécificateurs pour `printf` sont `%p` pour afficher une adresse, `%c` pour un caractère, `%fl` pour un double, `%lu` pour un long non signé (le retour de `sizeof`). Exemple :

   ```
   int v = 0;
   printf("%d %p\n", v, (void*) &v);
   ```

   Note : On doit utiliser un cast vers `void*` car `%p` ne permet d'afficher que les adresses génériques.

   Parfois, on a besoin de stocker un nombre de cases mémoires de manière contigües. Un tableau sera dès lors un tuple $(addr, size, n)$ où n est le nombre de cases mémoires de taille $size$ dont la première commence à l'adresse $addr$. On peut étendre les opérations définies précédement sur la mémoire pour prendre en compte ces tableaux :
   — Allocation dans la mémoire. Exemple : `int v[5];` va créer un tuple $(addr, sizeof(int), 5)$ dans la mémoire.

— Lire dans la mémoire. Exemple : `v[2]` va lire la valeur de v à l'adresse `&v + sizeof(int) * 2`. Par conséquent, on remarque que 2 est le décalage dans le tableau par rapport à l'adresse du tableau (multiplié par la taille de chacun d'un élément du tableau). Ainsi, les **tableaux sont indicés à partir de 0** en C, et on écrit donc `v[0]` pour accéder au premier élément.
— Écrire dans la mémoire. Exemple : `v[1] = 0;` va écrire la valeur 0 dans la case à l'adresse `&v + 1 * sizeof(int)` du tableau.

2. Un programme *average.c* utilisant les tableaux vous est donné sur le site web du cours. Lisez le et inspirez-vous en pour les exercices suivants.
3. Créer un tableau de 5 élements et compter le nombre de zéro.

   Pour les questions suivantes, sauf précisé, vous initialiserez un tableau de taille N avec des valeurs de votre choix—au lieu de les demander à l'utilisateur à chaque fois.

4. Comptez le nombre d'entiers x (définissez x vous même) présents dans le tableau.
5. Soit un tableau `numbers`, créer un tableau `accumulate` tel que `accumulate[i]` contient la somme des nombres de 0 à i de `numbers`. Écrivez cet algorithme avec une seule boucle.
6. Demandez à l'utilisateur une chaine de caractère et donner la longueur de la chaine entrée (une chaine est terminée par `'\0'`).
7. Afficher cette chaine à l'envers.

Exercice 3 – La fonction des fonctions

1. Faire une fonction `min` et une fonction `max` prenant 2 arguments que vous appellerez dans un programme demandant à l'utilisateur de rentrer 5 entiers et qui affichera le plus petit et le plus grand des 5.
2. Prenez le programme de l'exercice 3 du TD 2—intitulé "Mini-jeu"—et décomposez le en 5 fonctions dont voici les entêtes :

```
int ask_user_integer();
int generate_number();
// Return the result of the game
//    * `-1` if the guess is lower than the expected number,
//    * `0` if equal, and
//    * `1` if greater.
int compare_number(int expected, int guessed);
// Return 1 if he won, 0 otherwise.
int has_won(int game_result);
// The main loop putting all the functions together.
void process_loop();
```

3. Basé sur le même principe, créer un nouveau programme où c'est l'utilisateur qui dit le nombre à deviner et l'ordinateur qui joue. Est-ce que vous devez modifier beaucoup de fonctions ?
4. Reprenez les deux programmes précédents pour qu'ils utilisent le plus de code en commun.

À partir de maintenant, tous vos programmes ne pourront contenir que des fonctions de 5 lignes maximum (le `main` inclus). Et ce, pour tous les prochains TDs.

Exercice 4 – Le grand mix

Une difficulté des fonctions et tableaux est qu'une fonction doit pouvoir fonctionner avec des tableaux de toutes tailles. Imaginons qu'on écrive :

```
int sum(int numbers[50]) {
  ...
}
```

notre fonction ne marcherait qu'avec des tableaux avec une taille d'exactement 50 éléments... pas pratique. Par conséquent, on accompagne toujours un tableau avec sa taille dans les arguments d'une fonction :

```
int sum(int* numbers, int size) {
  ...
}

int v1[3] = { 1, 2, 3 };
int v2[2] = { 2, 5 };
int s1 = sum(v1, 3);
int s2 = sum(v2, 2);
```

C'est une grande différence par rapport aux autres langages, en C, un tableau ne connait pas sa taille ! Il faut donc la stocker dans une variable annexe. Pour ce qui est des tableaux de caractères c'est un peu différent puisqu'on sait qu'ils sont toujours terminés par un `'\0'`, donc on a pas besoin de la taille.

1. On peut constater qu'il y a un soucis avec `fgets` : le caractère de retour à la ligne `'\n'` est pris en compte. Créer une fonction `void read_line(char* input, int max)` qui lit une ligne de l'utilisateur et enlève le retour à la ligne problématique.
2. Faites une fonction qui vérifie que deux tableaux sont identiques.

Exercice 5 – Range ta chambre

Le tri à bulles ou tri par propagation est un algorithme de tri, qui consiste à permuter les éléments consécutifs non ordonnés d'un tableau. Il doit son nom au fait qu'il déplace rapidement les plus grands éléments en fin de tableau, comme des bulles d'air qui remonteraient rapidement à la surface d'un liquide. Ainsi après le premier passage, l'élément maximum est en dernière position du tableau et après k passages, les k derniers éléments du tableau sont ordonnés.

Exemple : à partir du tableau

3	2	1	6	5	4

Lors du premier passage on a :

2	3	1	6	5	4
2	1	3	6	5	4
2	1	3	5	6	4
2	1	3	5	4	6

et finalement l'élément 6 est bien placé. Il reste maintenant à trier le tableau contenant les 5 premiers éléments.

1. Proposez un algorithme permettant d'effectuer un tri à bulle, et évaluez en fonction du nombre d'éléments du tableau le nombre de tests et de permutations effectuées lors de ce tri.

Exercice 6 – Manipuler la mémoire

1. Soit un tableau `int t[5] = { 0 }`, afficher les adresses de chacunes des cases du tableau.
2. Quelle est l'adresse de la variable t ?
3. Déclarer une variable entière a et b juste avant et après `t`, quels sont leurs adresses ?
4. Pouvez-vous écrire dans a et b via la variable t (c'est-à-dire sans utiliser les noms a et b) ?

Exercice 7 – Challenge Time !

1. *11462–Age Sort* dont le pdf est disponible sur le site du cours. **Attention** : Vous devez trouver un algorithme qui trie linéairement en la taille de l'entrée (donc pas de boucle imbriquée).
2. *10252–Common Permutation* dont le pdf est disponible sur le site du cours. **Attention** : Il y a des tests supplémentaires dans le fichier *input-10252.txt*, vérifier que votre algorithme renvoie exactement la même chose que *output-10252.txt*.
3. *11475–Extend to Palindromes* dont le pdf est disponible sur le site du cours.