

# CRO TD3 :

Tableaux et structures de contrôles : if/switch , while/for, (goto), strings  
(1 séance sur machine)

## 1 Tableaux et boucle `for`

### Constructeurs de types

À partir des types prédéfinis du C (caractères, entiers, flottants), on peut créer de nouveaux types, appelés *types construits*, qui permettent de représenter des ensembles de données organisées. Ces *constructeurs de type* permettent de définir :

- Les tableaux grâce à l'opérateur ' [ ' ' ] '
- Les structures grâce au mot-clé `struct`
- Deux autres types construits qui nous verrons plus tard
  - Les unions avec le mot-clé `union`
  - Les énumérations avec le mot-clé `enum`
- Tous les autres types construits par des combinaisons de ces constructeurs de type : tableaux de tableaux, tableau de structure, unions de structure etc.

Un **tableau** est (plus ou moins dans tous les langages) une collection d'objets de même type, accessibles directement par un index. Les accès (lecture ou écriture) aux éléments de tableaux se font avec l'opérateur 'crochet', sauf pour l'initialisation lors de la déclaration. Par exemple si l'on considère un tableau de 5 valeurs entière contenant les 5 premiers entiers pairs, on le déclarera avec initialisation de la manière suivante en C :

```
int entierPair5[5]={2,4,6,8,10};
```

**Important :** En C les tableaux commencent toujours à l'indice 0, on accède au contenu d'un tableau `tab` à l'index `i` par l'opérateur crochet : `tab[i]`. Pour l'exemple ci-dessus on a donc `entierPair[0]` qui vaut 2, `entierPair[1]` qui vaut 4 etc. Évidemment le contenu des cases d'un tableau peuvent être modifiées au cours d'un programme.

**Important :** Autre convention importante en C : les case successives d'un tableau sont rangées de manière successive en mémoire, reprenez cela pour l'instant nous l'utiliserons plus tard.

On peut définir des tableaux multidimensionnels. Exemple : matrice `M` d'entiers de 10 lignes 5 colonnes `int M[10][5]`. Par convention encore, les éléments de la matrice sont rangés par ligne en mémoire.

QUESTION 1 ► Nous écrivons un programme avec un tableau lorsque nous aurons vu les boucles `for` (prochaine question). Combien d'élément et quel est l'index du dernier élément du tableau `tab` définit ainsi :

```
char tab[1024]
```

## Instruction et expression en C

En C il est important de comprendre que toutes les *expressions*, en C, ont un résultat et (éventuellement) un effet sur les variables en mémoire. Une *instruction* en revanche est une expression suivie d'un point-virgule, elle a donc un effet mais ne renvoi pas de résultat. Une des différence du C avec les autres langages est que même les affectations ont un résultat, ce résultat c'est l'évaluation de la partie droite de l'affectation. Si on écrit :

```
x = y = 2;
```

On aura bien `x` et `y` qui vaudront 2.








Plus formellement : Une *expression* est une suite de composants élémentaires syntaxiquement correcte, par exemple : `x = 0` ou bien `(i >= 0) && (i < 10)` qui produit :

- une action (modification possible de l'état des variables en mémoire)
- un résultat (valeur renvoyée par l'expression)

Une *instruction* est une expression suivie d'un point-virgule. Le point-virgule signifie en quelque sorte "évaluer cette expression et oublier le résultat". Plusieurs instructions peuvent être rassemblées par des accolades `{` et `}` pour former une *instruction composée* (ou *bloc*) qui est syntaxiquement équivalent à une instruction. Par exemple,

```
{  
  z = y / x;  
  t = y % x;  
}
```

**QUESTION 2** ► Remplissez, dans le tableau le type, le résultat et l'effet des expressions.

contexte	expression		
	<code>8*8.6</code>	type : résultat : 	effet :
<code>q</code> est de type <code>float</code>	<code>q - 2</code>	type : résultat : 	effet :
<code>q</code> est de type <code>int</code>	<code>p = (q++) + 1</code>	type : résultat : 	effet : 
<code>z</code> est de type <code>int</code>	<code>z == 1 ? 0 : 1</code>	type : résultat : 	effet :
<code>a</code> est de type <code>char</code> , <code>i</code> est de type <code>int</code>	<code>i = i + a</code>	type : résultat : 	effet : 

## Boucle `while` et boucle `for`

La syntaxe d'une boucle `while` en C est la suivante :

```
while (expression )  
  instruction
```

```
i = 1;  
while (i < 10)  
{  
  printf(" i = %d \n", i);  
  i++;  
}
```

**QUESTION 3** ► Écrivez une **fonction** qui vous demande répétitivement combien font `3*5`, et qui re-

tourne 15 lorsque vous avez enfin répondu la bonne réponse au clavier. testez votre fonction avec un `main` et un `Makefile`.

On évite d'utiliser les boucles `while`, sauf lorsque c'est indispensable, c'est à dire lorsque l'on ne connaît pas à l'avance le nombre d'itérations de la boucle. Lorsque l'on connaît le nombre d'itérations, on utilisera de préférence une boucle `for`.

### Boucle `for`

La syntaxe d'une boucle `for` en C est la suivante, la sémantique est expliquée par la boucle `while` équivalente.

```
for (expr 1 ; expr 2 ; expr 3)
    instruction
```

⇔

```
expr 1;
while (expr 2 )
{instruction
  expr 3;
}
```

Les boucles `for` sont donc des boucles `while` déguisées, on préférera quand même l'utilisation de boucles `for` quand cela est possible.

Attention à la gestion des indices : dans l'exemple ci-dessous, à la fin de cette boucle, `i` vaudra bien 10 :

```
for (i = 0; i < 10; i++)
    printf("\n i = %d", i);
```

Les trois expressions utilisées dans une boucle `for` peuvent être constituées de plusieurs expressions séparées par des virgules. Cela permet par exemple de faire plusieurs initialisations à la fois. Par exemple, pour calculer la factorielle d'un entier, on peut écrire (mais c'est à éviter) :

```
int n, i, fact;
for (i = 1, fact = 1; i <= n; i++)
    fact *= i;
printf("%d ! = %d \n", n, fact);
```

#### QUESTION 4 ► Écrivez une fonction :

```
int produit_scalaire(int A[N], int B[N])
```

qui calcule le produit scalaire de deux vecteurs entiers de taille `N`. On utilisera la directive `#define N 5` pour définir la valeur `N`. Testez votre fonction en initialisant les deux vecteurs et en affichant le résultat à l'écran.

## 2 Produit de matrice et entrées/sorties sur fichier

On va programmer un produit de matrices carrées. On rappelle que les coefficients de la matrice  $C = (c_{i,j})_{i,j}$ , produit des deux matrices carrées :  $A = (a_{i,j})_{i,j}$  et  $B = (b_{i,j})_{i,j}$  sont calculés avec la formule suivante :

$$\forall i, j \text{ t.q. } 1 \leq i, j \leq N, \quad c_{i,j} = \sum_{k=1}^{k=N} a_{i,k} * b_{k,j}$$

QUESTION 5 ► Téléchargez l'archive 'produit de matrice' sur Moodle, décompressez-la. Ces fichiers vont vous aider à tester votre programme de produit de matrice. parcourez les fichiers pour comprendre ce qu'ils font. Lisez l'encart suivant pour comprendre les entrées/sorties dans les fichiers.

## Entrée/Sortie fichier

En C, Les accès à un fichier se font par l'intermédiaire d'un tampon (buffer) qui permet de réduire le nombre d'accès au disque. Pour pouvoir manipuler un fichier, un programme a besoin d'un certain nombre d'informations : adresse du tampon, position dans le fichier, mode d'accès (lecture ou écriture) ... Ces informations sont rassemblées dans une structure dont le type, `FILE *`, est défini dans `stdio.h`. Un objet de type `FILE *` est un **flôt** (*stream*).

Avant de lire ou d'écrire dans un fichier, on l'*ouvre* par la commande `fich=fopen("nom-de-fichier", "r")`. Cette fonction dialogue avec le système d'exploitation et initialise un stream `fich`, qui sera ensuite utilisé lors de l'écriture ou de la lecture. Après les traitements, on *ferme* le fichier grâce à la fonction `fclose(fich)`.

Exemple : standard pour l'ouverture d'un fichier :

```
FILE *fich;
fich=fopen("monFichier.txt", "r");
if (!fich) fprintf(stderr, "Erreur d'ouverture : %s\n", "monFichier.txt");
```

Un objet de type `FILE *` est quelquefois appelé un descripteur de fichier, c'est un entier désignant quel est le fichier manipulé. Trois descripteurs de fichier existent **dans tous le programme C** et peuvent être utilisés sans qu'il soit nécessaire de les ouvrir (à condition d'utiliser `stdio.h`) :

- `stdin` (standard input) : unité d'entrée (par défaut, le clavier, valeur du descripteur : 1);
- `stdout` (standard output) : unité de sortie (par défaut, l'écran, valeur du descripteur : 0);
- `stderr` (standard error) : unité d'affichage des messages d'erreur (par défaut, l'écran, valeur du descripteur : 2).

Une fois ouvert en lecture (paramètre `"r"`) ou en écriture (paramètre `"w"`), on peut lire ou écrire dans un fichier avec les fonctions `fprintf` et `fscanf` exactement de la même manière que l'on lisait (au clavier) ou que l'on écrivait (à l'écran) avec les fonction `printf` ou `scanf`. En fait, `printf("hello World!\n")` est équivalent à `fprintf(stdout, "hello World!\n")`. la sortie standard est donc un fichier particulier.

Il existe d'autre fonctions pour lire ou écrire dans le fichiers lorsque les données ne sont pas formatable avec le format de `printf`.

**QUESTION 6** ► Exécutez la commande `make`, exécutez le programme produit. Comprenez vous ce qu'il se passe?

- étudiez les différents fichiers `.c` et `.h` ainsi que le `Makefile`, pourquoi la compilation du fichier `main.c` dépend-t-elle du fichier `matrice.h`? et `utilMatrice.h`?
- à quoi servent les macro `#ifndef` dans le fichier `matrice.h`?
- Pourquoi ces valeurs bizarres s'affichent lors de l'affichage du résultat `C=AB`.
- Pourquoi n'avons nous pas proposé une fonction qui retourne une matrice?
- Pourquoi peut-on modifier la matrice `C` avec la fonction `matProd` alors que l'on a vu que l'on passait les paramètres par valeur?
- Programmez la fonction `matProd` pour qu'elle fasse le produit de matrice.
- Ecrivez un nouvelle fonction `int matEqualQ(int A[N][N], int B[N][N])` qui vérifie que deux matrices sont égales.
- Validez votre fonction `matProd` en testant l'égalité avec la matrice solution proposée (`matriceAxB.txt`).

### 3 Instructions Conditionnelles : if et switch

#### Instructions Conditionnelles

Il n'y a pas de type booléen en C, on utilise le type entier pour cela : 0 veut dire False et n'importe quelle autre valeur veut dire True. La syntaxe des branchements conditionnels (if) en C est la suivante :

```
if (expression-1 )
    instruction-1
else instruction-2
```

ou

```
if (expression-1 )
    instruction-1
```

On peut, bien sûr, imbriquer les if-then-else et les if-then, la règle est que le else correspond au then le plus proche syntaxiquement. Attention à l'erreur classique :

if (a=1) {...} est une instruction correcte, mais qui ne fait pas la même chose que :  
if (a==1) {...} qui était probablement ce que le programmeur voulait écrire.

On peut aussi utiliser le branchement conditionnel multiple (switch) :

```
switch (expression )
{
    case constante-1:
        liste d'instructions 1
        break;
    case constante-2:
        liste d'instructions 2
        break;
    case constante-3:
        liste d'instructions 3
        break;
    default:
        liste d'instructions default
        break;
}
```

Attention, si l'on oublie les instructions break après une liste d'instruction, les instructions du cas suivant seront exécutées jusqu'à ce qu'on rencontre un break. C'est une des spécificité du langage C dont les motivations sont mystérieuses.

**QUESTION 7** ► Reprenez le corps de la fonction main.c fournie dans la question précédente. Comprenez vous maintenant la ligne suivante ?

```
if (!fichA)
{
    fprintf(stderr, "erreur d'ouverture du fichier %s\n", nomFichA);
    exit(-1);
}
```

La fonction fopen renvoi un entier (le descripteur de fichier) ou 0 lorsqu'il y a eu une erreur<sup>1</sup>, donc il y a un erreur si et seulement si fichA vaut 0. Dans tout les cas, on a bien if (!fichA) <=> if (fichA==0)

## 4 Pour aller plus loin : résolution de sudoku

### 4.1 Programmation : résolution de Su Doku

Une grille de Su Doku est une grille de taille  $9 \times 9$ , subdivisée en 9 carrés  $3 \times 3$  appelés *régions*. Les cases de cette grille peuvent recevoir des chiffres. Le jeu de Su Doku consiste à compléter une grille partiellement remplie par des chiffres en respectant la condition suivante : chaque ligne, colonne et région ne doit contenir qu'une seule fois tous les chiffres de un à neuf.

1. C'est un comportement qui est contraire a la convention presque systématique en C : En général une fonction renvoie un entier qui est interprété comme un code d'erreur, si elle renvoie 0 c'est qu'il n'y a pas d'erreur.

Un exemple de grille est représentée sur la gauche de la figure 1, la grille complétée est représentée sur la droite de la même figure. On peut vérifier que sur la grille de droite, chaque ligne contient tous les chiffres une et une seule fois, il en est de même pour chaque colonne et chaque région. Dans cette partie on utilisera un tableau bidimensionnel d'entiers de taille  $9 \times 9$  pour stocker une grille de Su Doku. Des fonctions vous sont proposées pour lire et écrire cette structure dans un fichier texte : téléchargez l'archive TP2.tar sur Moodle.

#### 4.1.1 Vérification de solution

- Écrire une fonction C qui vérifie qu'une grille complètement remplie est un Su Doku valide, la fonction renvoie 1 si le Su Doku est valide, 0 sinon. Voici le prototype de la fonction demandée :  

```
int sudokuValide(int sudoku[9][9])
```

#### 4.1.2 Résolution de Su Doku simples

Pour cette partie, nous allons supposer que nous n'avons que des grilles *simples* de Su Doku à résoudre (la définition de grille simple est donnée plus loin). On se propose d'implémenter la méthode suivante pour résoudre les Su Doku simples :

1. pour chaque chiffre  $i$  présent dans une case  $c$ , on mémorise le fait que ce chiffre  $i$  ne peut pas être mis dans une case présente sur la même ligne que celle de  $c$  ou sur même colonne que celle de  $c$  ou sur la même région que celle de  $c$ .
2. Compte-tenu de ces interdictions, on cherche une case vide pour laquelle il existe une *unique* valeur permise.
3. Si on ne trouve pas une telle case l'algorithme échoue (le Su Doku n'est pas *simple*).
4. Si on trouve une telle case : on la remplit, on mémorise les nouvelles interdictions introduites par le chiffre qu'on a rajouté et on repart en 2.

Un Su Doku sera donc dit *simple* si cette méthode permet de le résoudre, c'est à dire qu'à chaque étape il existe une case pour laquelle une seule valeur est possible. La grille de la figure 1 est un Su Doku simple : la figure 2 montre trois premières étapes possibles pour commencer le remplissage de la grille : la case de coordonnées (8,5) ne peut contenir que la valeur 1, ensuite la case de coordonnées (8,7) ne peut contenir que la valeur 5, ce qui entraîne que la case de coordonnées (7,7) ne peut contenir que la valeur 3 etc...

- Écrire une fonction C qui prend en entrée une grille partiellement remplie, et la remplit complètement avec cette méthode (ou échoue si le Su Doku n'est pas simple). Pour stocker les interdictions, on pourra par exemple utiliser un tableau d'entiers de dimension 3, chaque dimension étant de taille 9 :

```
int M[9][9][9]
```

La case  $M[i][j][k]$  vaudra 1 si le chiffre  $k+1$  ne peut pas être écrit dans la case de coordonnées  $(i+1, j+1)$  et 0 sinon (les tableaux en C commençant à 0, il faut ajouter 1 pour avoir des valeurs entre 1 et 9). Il n'y a pas obligation d'utiliser cette structure de données, vous pouvez proposer une autre structure de données pour écrire cette fonction.

		4		3	6			5
		5	1				8	
1	3	7	8		4	6		2
	5		4				3	6
4	9			7	3	2		8
		6		8		9	7	
8		9	7	6	5			1
6		2	3				4	
			9			8		7

9	8	4	2	3	6	7	1	5
2	6	5	1	9	7	4	8	3
1	3	7	8	5	4	6	9	2
7	5	8	4	2	9	1	3	6
4	9	1	6	7	3	2	5	8
3	2	6	5	8	1	9	7	4
8	4	9	7	6	5	3	2	1
6	7	2	3	1	8	5	4	9
5	1	3	9	4	2	8	6	7

FIGURE 1 – Une exemple de grille de Su Doku avec sa solution à droite

		4		3	6			5
		5	1				8	
1	3	7	8		4	6		2
	5		4				3	6
4	9			7	3	2		8
		6		8		9	7	
8		9	7	6	5			1
6		2	3	<b>1</b>			4	
			9			8		7

		4		3	6			5
		5	1				8	
1	3	7	8		4	6		2
	5		4				3	6
4	9			7	3	2		8
		6		8		9	7	
8		9	7	6	5			1
6		2	3	1		<b>5</b>	4	
			9			8		7

		4		3	6			5
		5	1				8	
1	3	7	8		4	6		2
	5		4				3	6
4	9			7	3	2		8
		6		8		9	7	
8		9	7	6	5	<b>3</b>		1
6		2	3	1		5	4	
			9			8		7

FIGURE 2 – Trois premières cases remplies par notre méthode (notez qu’il y a d’autres choix possibles à chaque étape)