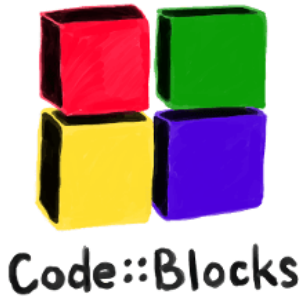


Extrait du référentiel : BTS Systèmes Numériques option A (Informatique et Réseaux)		Niveau(x)
S4. Développement logiciel S4.4. Programmation procédurale	Manipulations de données (« quoi ») en pseudo-langage et/ou en langage C Transcription d'algorithmes (« comment ») en pseudo-langage et/ou en langage C Développement de programmes « console » avec gestion des arguments de la ligne de commande	4 4 3

Objectifs du cours :

- Les tableaux simples (à une dimension)
- Les tableaux multidimensionnels
- Exercices

Les **tableaux** sont des regroupements de plusieurs objets. Ils regroupent des données de même type et de manière contiguë.

LES TABLEAUX SIMPLES (À UNE DIMENSION)

La définition d'un tableau nécessite trois informations :

- le type des éléments du tableau (un tableau est une suite de données de même type) ;
- le nom du tableau (son identificateur) ;
- la longueur du tableau (autrement dit, le nombre d'éléments qui le composent). Cette dernière doit être une expression entière.

La syntaxe de la déclaration est la suivante :

```
type identificateur[longueur];
```

La syntaxe de la déclaration d'un tableau est similaire à celle d'une variable, la seule différence étant qu'il est nécessaire de préciser le nombre d'éléments entre crochets à la suite de l'identificateur du tableau.

Ainsi, si nous souhaitons par exemple définir un tableau contenant vingt **int**, nous devons procéder comme suit.

Exemple :

```
int tab[20];
```

Comme pour les variables, il est possible d'initialiser un tableau ou, plus précisément, tout ou une partie de ses éléments. L'initialisation se réalise à l'aide d'une liste d'initialisation, séquentielle ou sélective.

Initialisation séquentielle avec une longueur explicite :

L'initialisation séquentielle permet de spécifier une valeur pour un ou plusieurs membres du tableau en partant du premier élément. Ainsi, l'exemple ci-dessous initialise les trois membres du tableau avec les valeurs 1, 2 et 3.

```
int tab[3] = { 1, 2, 3 };
```

Initialisation séquentielle avec une longueur implicite :

Lorsque vous initialisez un tableau, il vous est permis d'omettre la longueur de celui-ci, car le compilateur sera capable d'en déterminer la taille en comptant le nombre d'éléments présents dans la liste d'initialisation. Ainsi, l'exemple ci-dessous est correct et définit un tableau de trois **int** valant respectivement 1, 2 et 3.

```
int tab[] = { 1, 2, 3 };
```

Initialisation sélective avec une longueur explicite :

Il est possible de désigner spécifiquement les éléments du tableau que vous souhaitez initialiser. Cette initialisation sélective est réalisée à l'aide du numéro du ou des éléments.

Les éléments sont numérotés à partir de zéro.

L'exemple ci-dessous définit un tableau de trois **int** et initialise le troisième élément.

Exemple :

```
int tab[3] = { [2] = 3 };
```

Initialisation sélective avec une longueur implicite :

Dans le cas où la longueur du tableau n'est pas précisée, le compilateur déduira la taille du tableau du plus grand indice utilisé lors de l'initialisation sélective. Le code ci-dessous crée un tableau de cent **`int`**.

```
int tab[] = { [0] = 42, [1] = 64, [99] = 100 };
```

Dans le cas où vous ne fournissez pas un nombre suffisant de valeurs, les éléments oubliés seront initialisés à zéro ou, s'il s'agit de pointeurs, seront des pointeurs nuls.

Il est également possible de mélanger initialisations séquentielles et sélectives. Dans un tel cas, l'initialisation séquentielle reprend au dernier élément désigné par une initialisation sélective. Dès lors, le code ci-dessous définit un tableau de dix **`int`** et initialise le neuvième élément à 9 et le dixième à 10.

```
int tab[] = { [8] = 9, 10 };
```

Accès aux éléments d'un tableau :

L'accès aux éléments d'un tableau se réalise à l'aide d'un **indice**, un nombre entier correspondant à la position de chaque élément dans le tableau (premier, deuxième, troisième, etc). Cependant, il y a une petite subtilité : **les indices commencent toujours à zéro**. Ceci tient à une raison historique : les performances étant limitées à l'époque de la conception du langage C, il était impératif d'éviter des calculs inutiles, y compris lors de la compilation.

Plus précisément, l'accès aux différents éléments d'un tableau est réalisé à l'aide de **l'adresse de son premier élément**, à laquelle est ajouté l'indice. En effet, étant donné que tous les éléments ont la même taille et se suivent en mémoire, leurs adresses peuvent se calculer à l'aide de l'adresse du premier élément et d'un décalage par rapport à celle-ci (l'indice, donc).

Or, si le premier indice n'est pas zéro, mais par exemple un, cela signifie que le compilateur doit soustraire une unité à chaque indice lors de la compilation pour retrouver la bonne adresse, ce qui implique des calculs supplémentaires, chose impensable quand les ressources sont limitées.

Prenons un exemple avec un tableau composé de **`int`** (ayant une taille de quatre octets) et dont le premier élément est placé à l'adresse `0x0060FEF4`. Si vous déterminez à la main les adresses de chaque élément, vous obtiendrez ceci.

Indice	@ de l'élément
0	0x0060FEF4 (0060FEF4 + 0)
1	0x0060FEF8 (0060FEF4 + 4)
2	0x0060FEFC (0060FEF4 + 8)
3	0x0060FF00 (0060FEF4 + 12)
…	…

Il est possible de reformuler ceci à l'aide d'une multiplication entre l'indice et la taille d'un **`int`**.

Indice	@ de l'élément
0	0x0060FEF4 (0060FEF4 + (0x4))
1	0x0060FEF8 (0060FEF4 + (1x4))
2	0x0060FEFC (0060FEF4 + (2x4))
3	0x0060FF00 (0060FEF4 + (3x4))
…	…

Nous pouvons désormais formaliser mathématiquement tout ceci en posant **T** la taille d'un élément du tableau, **i** l'indice de cet élément, et **A** l'adresse de début du tableau (l'adresse du premier élément, donc). L'adresse de l'élément d'indice **i** s'obtient en calculant **A + i x T**.

Adresse du premier élément :

L'adresse du premier élément s'obtient en fait d'une manière plutôt contre-intuitive : lorsque vous utilisez une variable de type tableau dans une expression, celle-ci est convertie implicitement en un pointeur sur son premier élément. Comme vous pouvez le constater dans l'exemple qui suit, nous pouvons utiliser la variable **tab** comme nous l'aurions fait s'il s'agissait d'un pointeur.

Exemple :

```
#include <stdio.h>

int main(void)
{
    int tab[3] = { 1, 2, 3 };

    printf("Premier element : %d\n", *tab);
    return 0;
}
```

Résultat :

```
Premier element : 1
```

Il n'est pas possible d'affecter une valeur à une variable de type tableau, le code suivant est incorrect :

```
int t1[3];
int t2[3];
t1 = t2; /* Incorrect */
```

La règle de conversion implicite comprend néanmoins deux exceptions : l'opérateur **&** et l'opérateur **sizeof**.

Lorsqu'il est appliqué à une variable de type tableau, l'opérateur **&** produit comme résultat l'adresse du premier élément du tableau. Si vous exécutez le code ci-après, vous constaterez que les deux expressions donnent un résultat identique.

```
#include <stdio.h>

int main(void)
{
    int tab[3];

    printf("%p == %p\n", (void *)tab, (void *)&tab);
    return 0;
}
```

Exemple de résultat :

```
0060FEF4 == 0060FEF4
```

Dans le cas où une expression de type tableau est fournie comme opérande de l'opérateur **sizeof**, le résultat de celui-ci sera bien la taille totale du tableau et non la taille d'un pointeur.

```
#include <stdio.h>

int main(void)
{
    int tab[3];
    int *ptr;

    printf("sizeof tab = %zu\n", sizeof tab);
    printf("sizeof ptr = %zu\n", sizeof ptr);
    return 0;
}
```

Résultat :

```
sizeof tab = 12
sizeof ptr = 4
```

Cette propriété vous permet d'obtenir le nombre d'éléments d'un tableau à l'aide de l'expression suivante.

```
sizeof tab / sizeof tab[0]
```

Exemple pour obtenir la taille d'un type tableau :

```
#include <stdio.h>

int main(void)
{
    printf("sizeof int[3] = %zu\n", sizeof(int[3]));
    printf("sizeof double[42] = %zu\n", sizeof(double[42]));
    return 0;
}
```

Résultat :

sizeof int[3] = 12
sizeof double[42] = 336

Les autres éléments :

Pour accéder aux autres éléments, il va nous falloir ajouter la position de l'élément voulu à l'adresse du premier élément et ensuite utiliser l'adresse obtenue. Toutefois, recourir à la formule présentée au-dessus ne marchera pas car, en C, les pointeurs sont typés. Dès lors, lorsque vous additionnez un nombre à un pointeur, le compilateur multiplie automatiquement ce nombre par la taille du type d'objet référencé par le pointeur. Ainsi, pour un tableau de **int**, l'expression **tab + 1** est implicitement convertie en **tab + sizeof(int)**.

Exemple :

```
#include <stdio.h>

int main(void)
{
    int tab[3] = { 1, 5, 10 };

    printf("Premier element : %d\n", *tab);
    printf("Deuxieme element : %d\n", *(tab + 1));
    printf("Troisieme element : %d\n", *(tab + 2));
    return 0;
}
```

Résultat :

Premier element : 1
Deuxieme element : 5
Troisieme element : 10

L'expression ***(tab + i)** étant quelque peu lourde, il existe un opérateur plus concis pour réaliser cette opération : l'opérateur **[]**. Celui-ci s'utilise de la manière suivante : **expression[indice]** équivalent à ***(expression + indice).**

Exemple :

```
#include <stdio.h>

int main(void)
{
    int tab[3] = { 1, 5, 10 };

    printf("Premier element : %d\n", tab[0]);
    printf("Deuxieme element : %d\n", tab[1]);
    printf("Troisieme element : %d\n", tab[2]);
    return 0;
}
```

Résultat :

Identique au précédent.

Parcours et débordement :

Une des erreurs les plus fréquentes en C consiste à dépasser la taille d'un tableau, ce qui est appelé un cas de **débordement** (overflow en anglais). En effet, si vous tentez d'accéder à un objet qui ne fait pas partie de votre tableau, vous réalisez un accès mémoire non autorisé, ce qui provoquera un comportement indéfini. Cela arrive généralement lors d'un parcours de tableau à l'aide d'une boucle.

```
#include <stdio.h>

int main(void)
{
    int tableau[5] = { 784, 5, 45, -12001, 8 };
    int somme = 0;

    for (unsigned i = 0; i <= 5; ++i)
        somme += tableau[i];

    printf("%d\n", somme);
    return 0;
}
```

Le code ci-dessus est volontairement erroné et tente d'accéder à un élément qui se situe au-delà du tableau. Ceci provient de l'utilisation de l'opérateur `<=` à la place de l'opérateur `<` ce qui entraîne un tour de boucle avec `i` qui est égal à 5, alors que le dernier indice du tableau doit être quatre.

Tableaux et fonctions : passage en argument

Étant donné qu'un tableau peut être utilisé comme un pointeur sur son premier élément, lorsque vous passez un tableau en argument d'une fonction, celle-ci reçoit un pointeur vers le premier élément du tableau. Le plus souvent, il vous sera nécessaire de passer également la taille du tableau afin de pouvoir le parcourir.

Exemple : utilisation d'une fonction pour parcourir un tableau d'entiers et afficher les valeurs

```
#include <stdio.h>

void afficheTableau(int *tab, int taille)
{
    for (int i = 0; i < taille; ++i)
        printf("tab[%u] = %d\n", i, tab[i]);
}

int main(void)
{
    int tab[5] = { 2, 45, 67, 89, 123 };
```

```
    afficheTableau(tab, 5);
    return 0;
}
```

Résultat :

```
tab[0] = 2
tab[1] = 45
tab[2] = 67
tab[3] = 89
tab[4] = 123
```

Tableaux et fonctions : retour de fonction

De la même manière que pour le passage en argument, retourner un tableau revient à retourner un pointeur sur le premier élément de celui-ci. Toutefois, n'oubliez pas les problématiques de classe de stockage ! Si vous retournez un tableau de classe de stockage automatique, vous fournissez à la fonction appelante un pointeur vers un objet qui n'existe plus (puisque l'exécution de la fonction appelée est terminée).

```
#include <stdio.h>

int *tableau(void)
{
    int tab[5] = { 1, 2, 3, 4, 5 };

    return tab;
}

int main(void)
{
    int *p = tableau(); /* Incorrect */

    printf("%d\n", p[0]);
    return 0;
}
```

Résultat :

```
Process returned -1073741819 (0xC0000005)   execution time : 5.054 s
```

LES TABLEAUX MULTIDIMENSIONNELS

Jusqu'à présent, nous avons travaillé avec des tableaux linéaires, c'est-à-dire dont les éléments se suivaient les uns à la suite des autres. Il s'agit de tableaux dit à **une dimension** ou **unidimensionnels**.

Cependant, certaines données peuvent être représentées plus simplement sous la forme de tableaux à deux dimensions (autrement dit, organisées en lignes et en colonnes). C'est par exemple le cas des images (non vectorielles) qui sont des matrices de pixels ou, plus simplement, d'une grille de Sudoku qui est organisée en neuf lignes et en neuf colonnes.

Le langage C vous permet de créer et de gérer ce type de tableaux dit **multidimensionnels** (en fait, des tableaux de tableaux) et ce, bien au-delà de deux dimensions.

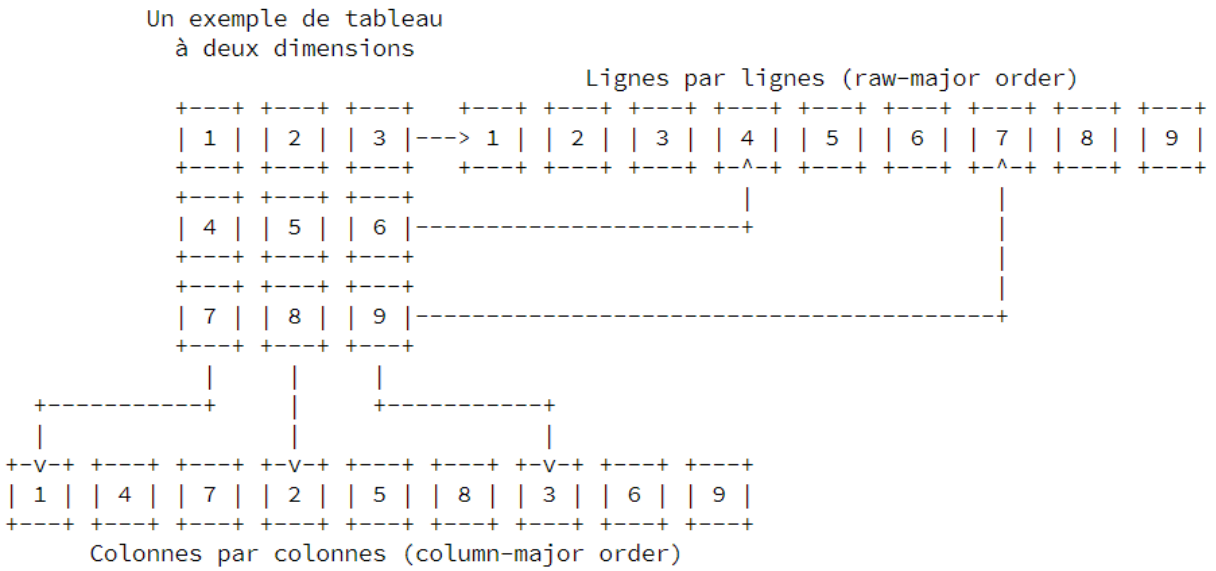
La définition d'un tableau multidimensionnel se réalise de la même manière que celle d'un tableau unidimensionnel si ce n'est que vous devez fournir la taille des différentes dimensions.

Par exemple, si vous sohaitez définir un tableau de **`int`** de vingt lignes et trente-cinq colonnes, vous procèderez comme suit.

```
int tab[20][35];
```

Représentation en mémoire :

Techniquement, les données d'un tableau multidimensionnel sont stockées les unes à côté des autres en mémoire : elles sont rassemblées dans un tableau à une seule dimension. Si les langages comme le FORTRAN mémorisent les colonnes les unes après les autres (column-major order en anglais), le C mémorise les tableaux lignes par lignes (row-major order).



Pour de plus amples informations, reportez-vous sur le document ressource associé à l'activité pratique sur le langage C : les tableaux.

EXERCICES

EXERCICE N°1

Question

Réalisez une fonction qui calcule la somme de tous les éléments d'un tableau de `int`.

EXERCICE N°2

Question

Créez deux fonctions : une qui retourne le plus petit élément d'un tableau de `int` et une qui renvoie le plus grand élément d'un tableau de `int`.