- Introduction à l'algorithmique et au langage C -

Tableaux multidimensionnels

TD8

1re année ESIEA - Semestre 1

L. Beaudoin & R. Erra & A. Gademer & L. Avanthey

2012 - 2013

Avant propos

Au cours des derniers TD & TP, nous nous sommes familiarisés avec la manipulation de tableaux unidimensionnels. Nous allons voir dans ce TD les possibilités que nous offre cette structure de données en terme de dimensions multiples.

1 Tableaux multidimensionnels : pourquoi faire ?

Une question qui vous vient peut-être à l'esprit est : « pourquoi une seule dimension ne suffit-elle pas ? ». Les tableaux unidimensionnels sont très pratiques et nous permettent de faire énormément de choses, mais ils ne peuvent pas répondre à tous nos problèmes. Par exemple si nous voulons mémoriser des informations concernant des positions dans l'espace, nous avons besoin de les identifier par deux coordonnées minimum (x et y), ce qui est intuitif avec un tableau bidimensionnel. Si nous rajoutons la troisième coordonnée de l'espace (z), un tableau tridimensionnel semble couler de source.

Bref, vous voyez où nous voulons en venir ? Dans de nombreux problèmes, nous aurons besoin de stocker des informations complexes multidimensionnelles : matrices, tableaux à n dimensions ou tableau de chaînes de caractères !

Les plus usités : 1D, 2D, 3D

Dans la majorité des cas notre utilisation se limitera à des tableaux tridimensionnels. Ce choix se justifie plus pour des raisons pratiques que techniques. En effet, le `langage C` en lui-même ne pose aucune contrainte, mais il est très difficile pour nous de représenter ou même d'imaginer les **hyperespaces** (plus de trois dimensions).

2 Déclaration de tableaux multidimensionnels

Nous avons vu qu'un tableau unidimensionnel se déclarait par son nom et son nombre de cases (taille) entre crochets. Pour un tableau multidimensionnel, c'est la même chose : nous le déclarons toujours par son **nom**, et nous précisons les **tailles** de chacune de ses dimensions **entre crochets**. Nous mettrons autant de paires de crochets qu'il y a de dimensions.

```
int matrix[2][3];
```

Tableaux 2D : lignes, colonnes ?

Les attributions des dimensions sont arbitraires et dépendent de vos choix personnels. Néanmoins, en `langage C`, il est d'usage de suivre le standard de la notation des matrices qui indique que la première dimension correspond aux lignes, et la seconde aux colonnes [1]. Selon ce standard, le tableau bidimensionnel que nous avons déclaré ci-dessus possède 2 lignes et 3 colonnes.

QUESTION 1

Sur le même principe, écrivez la déclaration des structures de données suivantes :
- Un tableau de flottants qui compte 20 lignes et 2 colonnes.
- Un tableau de caractères qui compte 4 lignes et 50 colonnes.
- Un tableau d'entiers permettant d'identifier les coordonnées des petits cubes d'un rubik's cube.
- Un tableau d'entiers de cinq dimensions. Toutes les dimensions de ce dernier possèdent la même taille : 30.

3 Accès aux éléments en lecture / écriture

Nous accédons aux emplacements mémoires exactement de la même manière qu'avec les tableaux unidimensionnels, c'est-à-dire avec le **nom** et les **indices** de chaque dimension **entre crochets**.

```c
/* Read access */
printf("La valeur de la case (1,2) est %d\n", matrix[1][2]);
/* Write access */
matrix[0][0] = 1;
```

QUESTION 2

Sur le même principe écrivez les instructions pour réaliser les actions suivantes :
- Afficher la case d'indice (3,4) du tableau d'entier `array`.
- Afficher la lettre qui se trouve à la dixième ligne et à la septième colonne du tableau `array`.
- Affecter la valeur approchée de PI à la case d'indices (3, 1, 4) du tableau `array`  .

4 Algorithmes de parcours

Nous avons vu que le parcours de tous les éléments d'un tableau unidimensionnel se fait à l'aide d'**une boucle itérative** sur les indices qui permet de visiter chaque case.

Dans le cas des tableaux multidimensionnels, chaque case est identifiée par plusieurs indices. Pour accéder à toutes les cases, il va donc falloir traiter toutes les combinaisons possibles d'indices. Le plus simple est de parcourir une dimension pour commencer (avec une première boucle), puis pour chaque indice de cette dimension parcourir la deuxième dimension (avec une deuxième boucle imbriquée), puis pour chaque indice de celle-ci parcourir la troisième dimension (une troisième boucle imbriquée) et ainsi de suite. Autrement dit, il nous faudra **autant de boucles imbriquées qu'il y a de dimensions**.

Par exemple pour un tableau bidimensionnel, nous devons parcourir toutes les lignes, puis pour chaque ligne, parcourir toutes les colonnes.

QUESTION 3

En vous inspirant de celui que nous avons vu pour les tableaux unidimensionnels, écrivez l'algorithme de parcours d'un tableau bidimensionnel.

1. Appelé « Row-major order ».

Bien nommer ses itérateurs

Pour éviter de vous perdre quand vous manipulez plusieurs dimensions, prenez l'habitude de nommer vos dimensions de manière significative. Par exemple pour un tableau de deux dimensions, appelez-les *lin* et *col* (pour les tailles de dimensions associées : *nbLin* ou *height* et *nbCol* ou *width*).

QUESTION 4

Toujours sur le même principe, écrivez l'algorithme de parcours d'un tableau tridimensionnel.

4.1 Initialisation

Nous avions vu que la méthode principale pour initialiser un tableau était de parcourir toutes ses cases et de leur affecter une valeur à chacune. Cela ne change pas quelque soit le nombre de dimensions du tableau, il suffit juste d'utiliser l'algorithme de parcours adapté.

QUESTION 5

Soit un tableau d'entiers `iArray` de 100 lignes et 200 colonnes. Déclarez ce tableau, puis **en utilisant votre algorithme de parcours**, écrivez le code pour initialiser toutes ses cases à 0.

QUESTION 6

Soit un tableau de flottants tridimensionnel (50, 100, 100). Toujours **en utilisant votre algorithme de parcours**, écrivez le code pour initialiser toutes les cases à 1.0.

Quand nous déclarons de très petits tableaux fixes ou que nous avons besoin de faire des tests, nous pouvons également utiliser la notation accolades :

```
int matrix[2][3] = { { 1, 2, 3 }, { 4, 5, 6 } };
```

Notez le cas particulier qui permet d'initialiser un tableau à 0 à la déclaration (ce **n'est pas valable** pour des nombres autres que 0) :

```
int matrix[200][300] = { { 0 }, { 0 } };
```

5 Tableaux multidimensionnels et fonctions

5.1 Tableaux multidimensionnels en paramètres

En paramètres, comme pour les tableaux unidimensionnels, les tableaux multidimensionnels se déclarent avec leur type, leur nom et autant de paires de crochets qu'il y a de dimensions. Nous passons également, à l'aide d'autres paramètres les informations concernant les tailles de chacune des dimensions.

Pour le cas des tableaux multidimensionnels, la différence avec les tableaux unidimensionnels vient du fait que **nous ne pouvons pas laisser les crochets vides lors de la déclaration du paramètre**.

Dans la mesure où nous voulons obtenir des fonctions génériques (c'est-à-dire qui fonctionnent quelques soient les tailles des tableaux), nous fixerons ces dimensions grâce aux tailles passées en paramètres. Attention à l'ordre des paramètres : les tailles doivent absolument être placées **AVANT** la variable tableau.

```
void myFunction(int nbLin, int nbCol, int iArray[nbLin][nbCol]);
void myFunction1(int nbDim1, int nbDim2, int nbDim3, int iArray[nbDim1][nbDim2][nbDim3]);
```

Remarque

Vous trouverez parfois dans la littérature une notation où la taille de la première dimension n'est pas fixée dans les crochets : `void myFunction(int nbLin, int nbCol, int iArray[][nbCol]);`. Cette notation est autorisée mais nous vous la **déconseillons**.

5.2 Tableaux multidimensionnels en arguments

En argument, quelque soit le nombre de dimensions, nous passons toujours un tableau avec son nom uniquement (sans les crochets). Seules les informations que nous envoyons à la fonction sur la taille du tableau changent. Nous devons donc envoyer autant de tailles qu'il a de dimensions.

```
int myArray2D[10][2];
int myArray3D[2][3][4];
myFunction2DArray(10, 2, myArray2D);
myFunction3DArray(2, 3, 4, myArray3D);
```

5.3 Un peu de pratique

EXERCICE 1

Créez un fichier et déclarez dans le `main` un tableau d'entiers du nombre de lignes et de colonnes que vous souhaitez (environ 4/6 de chaque) qui vous servira à tester vos fonctions.

EXERCICE 2

En **utilisant l'algorithme de parcours**, écrivez la fonction `void initArray(int nbLin, int nbCol, int iArray[nbLin][nbCol])` qui permet d'initialiser un tableau bidimensionnel avec les valeurs de 1 à nombre de cases. Voici un exemple sur un tableau (4,5) :

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

EXERCICE 3

Toujours **en utilisant l'algorithme de parcours**, écrivez la fonction d'affichage `void showArray (int nbLin, int nbCol, int iArray[nbLin][nbCol])` qui permet d'afficher les éléments d'un tableau bidimensionnel. Vous séparerez chaque élément par une espace, et vous terminerez chaque ligne par un retour à la ligne. Pour garder l'alignement des colonnes, vous afficherez automatiquement les valeurs sur une largeur de trois caractères grâce au descripteur suivant :`"%3d"`.

EXERCICE 4

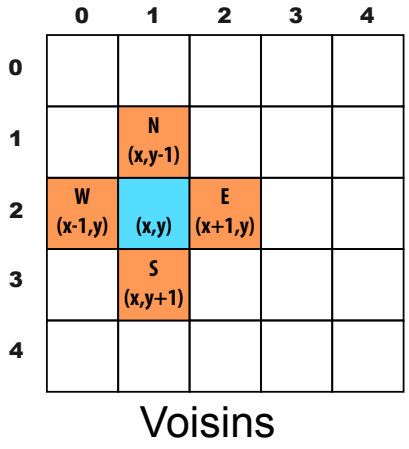
Rajoutez l'appel de ces deux premières fonctions dans le `main` et testez votre programme.

EXERCICE 5

Toujours en **utilisant l'algorithme de parcours**, écrivez la fonction `void loadArray(int nbLin , int nbCol, int iArray[nbLin][nbCol])` qui permet d'initialiser un tableau bidimensionnel avec des valeurs entrées aux claviers. Testez cette fonction dans votre `main`.

EXERCICE 6

Écrivez la fonction `int getSumOfFourNeighbors(int nbLin, int nbCol, int iArray[nbLin][nbCol], int eltLin, int eltCol)` qui doit calculer et retourner la somme des 4 voisins adjacents de la case donnée lorsqu'ils existent.



Attention aux bords !

Quand vous êtes sur les bords, certains de vos voisins n'existent pas. N'essayez donc pas d'accéder à des cases en dehors du tableau !

Par exemple dans le tableau donné, si nous demandons la somme des 4 voisins de la case d'indices (1,1), nous obtenons 28 ($2 + 6 + 8 + 12$). Mais dans le cas où nous nous trouvons sur un bord, la case (1,0) par exemple, nous ne prenons en compte que les cases qui existent, évidement. Nous obtenons alors 19 ($1 + 7 + 11$). Un voisin sur quatre n'existe pas, il ne faut pas essayer d'aller sur cette case fantôme, sinon nous sortons du tableau.

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

EXERCICE 7

Rajoutez l'appel de cette fonction dans le `main` et testez votre programme. N'oubliez pas de tester différents types de case : au milieu, sur le bord gauche, sur le bord droit, en haut, en bas, dans un coin, etc..

EXERCICE 8

Écrivez la fonction `int getSumOfEightNeighbors(int nbLin, int nbCol, int iArray[nbLin][nbCol], int eltLin, int eltCol)` qui doit calculer la somme des 8 voisins adjacents à la case donnée lorsqu'ils existent. Par rapport à la fonction précédente, nous ajoutons les 4 cases en diagonales.

EXERCICE 9

Rajoutez l'appel de cette fonction dans le `main` et testez votre programme. N'oubliez pas de tester différents types de case : au milieu, sur le bord gauche, sur le bord droit, en haut, en bas, dans un coin, etc..

Mémo

« Pour déclarer un tableau multidimensionnel, nous ajoutons autant de paires de crochets qu'il y a de dimensions. »

« En paramètre, il est nécessaire de fixer les dimensions, en passant les tailles AVANT le tableau. »