

Travaux Pratiques thème 4 : Les pointeurs et les tableaux dynamiques

Pointeurs

Exercice 1

Ecrire une fonction `incrimente` qui prend 3 paramètres en entrée (**a**, **b** et **i**), et qui ajoute à **a** et **b** la valeur **i**. Cette fonction retourne la valeur 1 si **i** vaut 0, 0 sinon.

Exercice 2

Ecrire le code C permettant de créer un tableau **tab** de 17 entiers en utilisant un pointeur et la fonction `malloc`. Libérer ensuite la mémoire allouée au tableau avec la fonction `free`.

Exercice 3

Ecrire le code C permettant de créer un tableau **tab** à deux dimensions de 15 lignes et 17 colonnes entiers en utilisant un pointeur et la fonction `malloc`. Libérer ensuite la mémoire allouée au tableau avec la fonction `free`.

Tableaux et Matrices

Exercice 4

- Ecrire une fonction qui demande à l'utilisateur le nombre de lignes et de colonnes d'une matrices d'entiers, puis remplit la matrice en demandant les valeurs à l'utilisateur.
- Ecrire une fonction `affiche_mat(m, l, c)` qui permet d'afficher une matrice m de l lignes et c colonnes.

Exercice 5

Ecrire une fonction permettant de multiplier deux matrices. On n'oubliera pas de traiter les cas d'erreur.

Exercice 6

Le but de cet exercice est de calculer le déterminant d'une matrice carrée $n \times n$.

1. Programmer une fonction qui prends une matrice carrée ($n > 1$) en entrée et deux entiers i et j et qui renvoie la même matrice privée de la ligne i et de la colonne j .

2. Programmer la fonction **det** en utilisant la formule suivante :

$$\det(M) = \sum_{i=1}^n (m_{ij} \times \text{Cof}_{i,j}) \quad (1)$$

dans laquelle m_{ij} est le terme de ligne i et de colonne j de la matrice M , et $\text{Cof}_{i,j}$ est égal à -1^{i+j} multiplié par le déterminant de la matrice M privée de la ligne i et de la colonne j .

On notera que le déterminant d'une matrice M de taille 1×1 est égale à la valeur contenue dans la matrice.

Labyrinthe, application des tableaux 2D

Exercice 7

On souhaite réaliser un programme qui recherche un chemin menant de l'entrée à la sortie au travers d'un labyrinthe. Le labyrinthe est représenté par une matrice carrée $\text{DIM} \times \text{DIM}$, contenant des 0 et des 1. Un 0 en position (i, j) indique que la case (i, j) ne contient pas d'obstacle, un 1 représente un obstacle. On utilisera un tableau 2D pour stocker la matrice. On souhaite dans un premier temps afficher le labyrinthe, en représentant une case inoccupée par un caractère ESPACE et un obstacle par le caractère ÉTOILE. Pour se faire, on définit un tableau de caractères (de nom palette) dont l'indice désigne le statut de la case du labyrinthe (libre=0 ou occupé=1) et le contenu le caractère à afficher pour ce statut (ESPACE ou ÉTOILE).

Soit la déclaration d'une telle palette de *couleurs* :

```
char palette[2] = ' ', '*';
```

La procédure **afficher_lab** prend en argument le labyrinthe et la palette, et affiche le contenu de la matrice en fonction des caractères associés dans la palette.

Pour le tableau suivant :

0	0	1	0	0	0	1	1
1	0	0	1	1	0	1	0
1	0	1	1	1	0	0	0
1	0	0	1	0	0	1	1
0	1	0	0	0	1	0	1
0	0	0	1	0	1	0	1
1	1	1	1	0	0	0	0
1	0	1	1	0	0	0	0

Voilà le résultat que vous devez obtenir :

```
+-----+
|  *    ** |
| *  ** *  |
| * ***    |
| *  *    ** |
| *    *  * |
|    *  *  * |
| ****      |
| *  **      |
+-----+
```

1. Ecrivez la fonction :

```
void afficher_lab(int lab[DIM][DIM], char palette[])
```

Le paramètre formel **lab** correspond au labyrinthe passé en argument.

L'algorithme de recherche d'un chemin menant de l'entrée vers la sortie du labyrinthe est intrinsèquement récursif : à partir d'un point (i, j) non occupé (qui est l'extrémité du chemin en cours de construction), on recherche si le chemin peut être poursuivi, c'est à dire si l'un des voisins de (i, j) est également inoccupé. Si c'est le cas, le chemin progresse vers ce voisin inoccupé et la même recherche se poursuit à partir de ce point (i', j') , jusqu'à ce que la sortie soit atteinte. Si aucun voisin de (i, j) ne peut poursuivre

le chemin, alors (i, j) n'est pas sur le chemin menant de l'entrée à la sortie. On distingue ainsi quatre états possibles pour une position du labyrinthe en cours d'examen :

0 : espace inoccupé non encore atteint par la recherche du chemin,

1 : espace occupé

2 : espace inoccupé en cours d'analyse (il est sur le chemin en cours de construction)

3 : espace inoccupé dont on est sûr qu'il n'est pas sur le chemin.

Par convention, le point de départ sera toujours $(0, 0)$ et le point d'arrivée $(m - 1, n - 1)$.

Pour le labyrinthe donné précédemment, la recherche de chemin donnera les étapes suivantes :

Départ en $(0, 0)$:

2	0	1	0	0	0	1	1
1	0	0	1	1	0	1	0
1	0	1	1	1	0	0	0
1	0	0	1	0	0	1	1
0	1	0	0	0	1	0	1
0	0	0	1	0	1	0	1
1	1	1	1	0	0	0	0
1	0	1	1	0	0	0	0

L'étape suivante considèrera une case voisine de $(0, 0)$, par exemple la case à droite $(0, 1)$. L'appel sur les autres cases voisines de $(0, 0)$ sera réalisé après avoir examiné les possibilités de chemin offertes par $(0, 1)$ et uniquement si aucune n'a abouti.

Case $(0, 1)$:

2	2	1	0	0	0	1	1
1	0	0	1	1	0	1	0
1	0	1	1	1	0	0	0
1	0	0	1	0	0	1	1
0	1	0	0	0	1	0	1
0	0	0	1	0	1	0	1
1	1	1	1	0	0	0	0
1	0	1	1	0	0	0	0

L'appel récursif pour la cases à droite échoue (obstacle), on continue donc avec une autre case voisine, par exemple celle qui est en dessous. Comme précédemment, les autres possibilités ne seront examinées que si aucun chemin n'est trouvé en examinant cette case.

Case $(1, 1)$:

2	2	1	0	0	0	1	1
1	2	0	1	1	0	1	0
1	0	1	1	1	0	0	0
1	0	0	1	0	0	1	1
0	1	0	0	0	1	0	1
0	0	0	1	0	1	0	1
1	1	1	1	0	0	0	0
1	0	1	1	0	0	0	0

On examine la case de droite.

Case $(1, 2)$:

2	2	1	0	0	0	1	1
1	2	2	1	1	0	1	0
1	0	1	1	1	0	0	0
1	0	0	1	0	0	1	1
0	1	0	0	0	1	0	1
0	0	0	1	0	1	0	1
1	1	1	1	0	0	0	0
1	0	1	1	0	0	0	0

Aucune possibilité à partir de cette case :

- case droite : obstacle
- case au-dessous : obstacle
- case gauche : déjà visitée
- case au-dessus : obstacle

L'examen de la case (1,2) est donc terminé, aucun chemin permettant d'accéder à la sortie a été trouvé, cette case est marquée avec un 3 et on reprend l'étape laissée en suspens le plus récemment, à savoir la case (2,1), case située en dessous de la case (1,1), et ainsi de suite.

A la fin, le tableau ressemblera à ça :

2	2	1	0	0	0	1	1
1	2	3	1	1	0	1	0
1	2	1	1	1	0	0	0
1	2	2	1	0	0	1	1
0	1	2	2	2	1	0	1
0	0	0	1	2	1	0	1
1	1	1	1	2	2	2	2
1	0	1	1	0	0	0	2

Il est à noter que le chemin final dépend de l'ordre dans lequel les cases sont examinées (l'ordre utilisé ici était droite, bas, gauche, haut). Quel que soit l'ordre, si un chemin est possible il doit être trouvé, par contre si plusieurs chemins sont possibles, celui qui sera trouvé dépendra de l'ordre utilisé. De même toutes les cases ne sont pas nécessairement explorées.

2. Programmez la procédure de recherche d'un chemin dans le labyrinthe :

```
int chercher_chemin(int lab[DIM][DIM])
```

Cette fonction retourne un entier correspondant à un booléen représentant le résultat de la recherche. Si aucun chemin n'a été trouvé, la fonction retournera 0. Cette fonction pourra faire appel à la fonction récursive suivante pour la recherche de chemin :

```
int chercher_chemin_rec(int lab[DIM][DIM], int lc, int cc)
```

Les arguments `lc` et `cc` correspondent à la position courante lors de la recherche du chemin. Cette fonction retourne 1 si un chemin a été trouvé, 0 sinon. Vous pourrez insérer des instructions d'affichage du tableau 2D en cours de recherche de chemin pour visualiser le déroulement de l'algorithme.

3. Écrivez maintenant un programme de test.