SÉANCE 6

Objectif

Le but de cette sixième séance est de poursuivre le travail sur la gestion dynamique de la mémoire en l'appliquant aux tableaux à deux dimensions.

1 Introduction

Dans l'exercice 2 de la séance 3, nous avons manipulé des tableaux à deux dimensions (matrices) statiques. Nous avons été obligés de faire une hypothèse sur la taille maximale d'une matrice. Il s'agissait de matrices carrées dont l'ordre était supposé inférieur ou égal à 10. Dans le cas général d'un tableau à deux dimensions, avec une gestion statique de la mémoire, on doit faire des hypothèses sur le nombre maximal de lignes et le nombre maximal de colonnes. Par exemple, pour une matrice contenant des valeurs de types `unsigned char`, on peut écrire :

```
#define NBLIGMAX 100
#define NBCOLMAX 100
unsigned char tab[NBLIGMAX][NBCOLMAX];
```

Cela permet bien sûr d'accéder aux éléments en utilisant la syntaxe `tab[i][j]`.

Si, au moment de l'exécution, l'utilisateur n'a besoin que d'une matrice de 2 lignes et 3 colonnes, nous avons réservé beaucoup de place en mémoire (9994 octets) qui ne sera pas utilisée. La gestion dynamique de la mémoire permet de mieux l'utiliser.

2 Matrices dynamiques

Il y a principalement deux possibilités pour gérer de manière dynamique les tableaux à deux dimensions : en utilisant un tableau dynamique à une dimension ou deux tableaux dynamiques à une dimension.

▶ *Un tableau dynamique à une dimension*

- Déclaration : `unsigned char *tab;`
- Allocation dynamique : `tab=malloc(NbLig*NbCol*sizeof(unsigned char));`
  `NbLig` et `NbCol` peuvent être des variables.
- Accès aux éléments : `tab[i*NbCol+j]`

▶ *Deux tableaux dynamiques à une dimension*

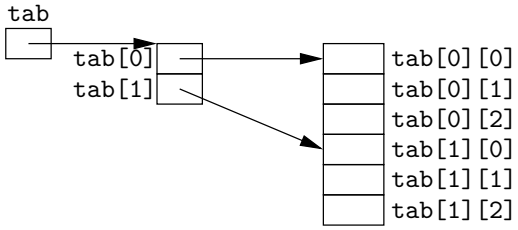
Un tableau contient les éléments de la matrice et l'autre contient les adresses du premier élément de chaque ligne. On rappelle que :

```
tab[i][j]==(tab[i])[j]
          ==*(tab[i]+j)
          ==*(*(tab+i)+j)
```

On peut donc considérer que `tab` est un tableau de pointeurs vers le début de chaque ligne.

- Déclaration : `unsigned char **tab;`
- Accès aux éléments : `tab[i][j]`

Exemple : si `tab` contient 2 lignes et 3 colonnes :



Cette technique permet d'accéder aux éléments en évitant la multiplication et en utilisant les deux couples de crochets (comme avec un tableau statique).

Une variante consiste à allouer séparément les différentes lignes. Dans ce cas, la structure de données est constituée du tableau de pointeurs vers le début de chaque ligne et de `NbLig` tableaux constituant les différentes lignes. Elle a pour inconvénient de ne pas avoir tous les éléments du tableau les uns à la suite des autres en mémoire. Mais elle a l'avantage, dans le cas de tableaux de grande taille, d'éviter de devoir disposer d'un bloc de taille suffisante disponible en mémoire.

Exercices

C'est la deuxième possibilité décrite ci-dessus, c'est-à-dire l'utilisation de deux tableaux dynamiques à une dimension, mais pas sa variante, que l'on va implémenter. On définit pour cela le synonyme de type suivant :

```
typedef unsigned char **Matrice;
```

qui va servir pour manipuler les matrices.

L'objectif de ces exercices est d'écrire des fonctions permettant de gérer des matrices dynamiques selon le principe décrit ci-dessus. Testez ces fonctions en écrivant une fonction principale.

Exercice 1

Écrire une fonction d'en-tête :

```
Matrice MatAllouer(int NbLig, int NbCol)
```

qui alloue et initialise l'espace mémoire nécessaire pour stocker une matrice contenant `NbLig` lignes et `NbCol` colonnes. Cette fonction doit retourner la matrice ou `NULL` en cas de problème. Ses principales étapes sont :

- l'allocation du tableau contenant les éléments ;
- l'allocation du tableau des pointeurs ;
- le remplissage du tableau des pointeurs avec les adresses de début des lignes ;
- la transmission à l'appelant de l'accès à la matrice.

Exercice 2

Écrire une fonction d'en-tête :

```
Matrice MatLire(int *pNbLig, int *pNbCol)
```

qui :

- demande à l'utilisateur le nombre de lignes et le nombre de colonnes de la matrice qu'il souhaite saisir ;
- alloue la matrice grâce à l'appel de la fonction précédente ;
- lit, ligne par ligne (la lecture d'une valeur de type `unsigned char` nécessite d'utiliser le spécificateur de format `%hhu`), les éléments et les stocke dans la matrice ;
- délivre en sortie le nombre de lignes et de colonnes de la matrice ;
- retourne la matrice ou `NULL` en cas de problème.

☞ Exercice 3

Écrire une fonction d'en-tête :

```
void MatAfficher(Matrice Mat, int NbLig, int NbCol)
```

qui affiche à l'écran le contenu de la matrice passée en paramètre (l'affichage d'une valeur de type `unsigned char` se fait avec le spécificateur de format `%d`).

☞ Exercice 4

Écrire une fonction d'en-tête :

```
Matrice MatCopier(Matrice Mat, int NbLig, int NbCol)
```

qui réalise une copie de la matrice `Mat` de taille `NbLig`×`NbCol`. Cette fonction doit retourner la copie ou `NULL` en cas de problème. Ses principales étapes sont :

- l'allocation de la copie ;
- la copie des éléments ;
- la transmission à l'appelant de l'accès à la copie.

☞ Exercice 5

Écrire une fonction d'en-tête :

```
void MatLiberer(Matrice *pMat)
```

qui libère tout l'espace mémoire occupé par la matrice d'adresse `pMat` (le tableau contenant les éléments et le tableau des pointeurs). Cette fonction doit aussi affecter la valeur `NULL` à la matrice libérée (ceci permet d'éviter les erreurs à l'exécution si on essaie de libérer plusieurs fois la même matrice).