Développement C **L2 CUPGE 2018–2019**

Travaux pratiques Matrices

Le but de ce TP est d'explorer les tableaux dynamiques en 2 dimensions.

I Exercice : *Matrices statiques*

On commence par utiliser des matrices dont les tailles sont fixées. Définissez deux constantes `NB_LIGNES` et `NB_COLONNES` valant 2.

Solution

```
#define NB_LIGNES 2
#define NB_COLONNES 2
```

(a) Définissez le type `matrice` comme étant le type des tableaux de `float` ayant `NB_LIGNES` lignes et `NB_COLONNES` colonnes.

Solution

```
typedef float matrice[NB_LIGNES][NB_COLONNES] ;
```

(b) Écrivez (et testez dans la fonction `main`) une fonction qui affiche une matrice passée en argument.

Solution

```
void affiche_matrice ( matrice M )
{
  int i , j ;
  for ( i = 0 ; i < NB_LIGNES; i++ )
  {
    printf ( "(\t%f\t" , M[i][0] ) ;
    for ( j = 1 ; j < NB_COLONNES ; j++ )
      printf ( "%f\t" , M[i][j] ) ;
    printf ( ")\n''␣;
␣␣}
␣␣printf␣(␣"\n"␣)␣;
}
␣␣
```

(c) Écrivez (et testez dans la fonction `main`) une fonction qui demande à l'utilisateur les coefficients d'une matrice passée en argument. Comme on utilise la librairie `libentrees.a` et les macros de test, créez aussi le `Makefile` suivant :

```
exo1 : exo1.c
	gcc -Wall -o exo1 exo1.c -I ../../include -L ../../lib -lentrees
```

Solution

```
void lit_matrice ( matrice M )
{
  int i , j ;
  for ( i = 0 ; i < NB_LIGNES; i++ )
  {
    printf ( "Coefficients de la ligne %d :\n" , i + 1 ) ;
    for ( j = 0 ; j < NB_COLONNES ; j++ )
    {
      printf ( "M[%d][%d] = " , i + 1 , j + 1 ) ;
      LireDecimal ( M[i]+j ) ;
    }
  }
}
```

(d) Dans la fonction `main`, lisez 2 matrices m_1 et m_2 données par l'utilisateur et affichez-les.

(e) Créez une fonction `somme_matrices`, et calculez la somme m_3 de $m_1 + m_2$, et affichez le résultat.

Solution

```
void somme_matrices ( matrice res , matrice a , matrice b )
{
  int i , j ;
  for ( i = 0 ; i < NB_LIGNES ; i++ )
    for ( j = 0 ; j < NB_COLONNES ; j++ )
      res[i][j] = a[i][j] + b[i][j] ;
}
```

(f) Créez une fonction `produit_matrices`, calculez le produit m_3 de $m_1 \times m_2$, et affichez le résultat.

Solution

```
void produit_matrices ( matrice res , matrice a , matrice b )
{
  int i , j ;
  for ( i = 0 ; i < NB_LIGNES ; i++ )
    for ( j = 0 ; j < NB_COLONNES ; j++ )
    {
      res[i][j] = 0 ;
      for ( k = 0 ; k < NB_LIGNES ; k++ )
        res[i][j] += a[i][k] * b[k][j] ;
```

```
    }
}
```

II Exercice : *Matrices de taille variable*

Avoir des matrices qui ont toutes une taille fixée n'est pas très utile. On va utiliser des structures et l'allocation dynamique avec `malloc` pour avoir des matrices avec un nombre variable de lignes et de colonnes.

Avant de commencer, dans l'exercice précédent, si on analyse la définition d'une matrice `M`, on constate que :

- `M[0]` est un tableau de 2 `float` ;
- `M[1]` est un tableau de 2 `float` ;
- la valeur de `M[0]` est égale à la valeur de `M` (l'adresse du début de la matrice), et est l'adresse de la case `M[0][0]` ;
- la valeur de `M[1]` (l'adresse de début du second tableau) est aussi la valeur de l'adresse de `M[1][0]` ;
- d'après l'arithmétique des pointeurs, on a, pour toute case x de type t :

$$\&*x = x \tag{1}$$
$$(\texttt{int})(\&x+1) = ((\text{int})\&x) + \texttt{sizeof}(t) \tag{2}$$
$$\tag{3}$$

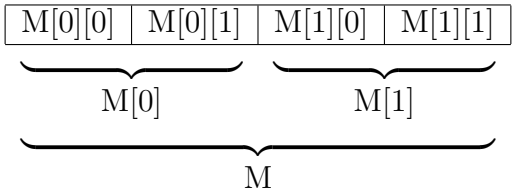
Donc :

case	type	taille
M[0][0]	float	sizeof (float)
M[0]	float [2]	2 * sizeof (float)
M	float[2][2]	2*(2 * sizeof (float))

Et donc :

$$(\text{ int })(\&M+1) = ((\text{ int })\&M) + 4 \texttt{ * sizeof (float)}$$
La matrice juste après la fin de M
$$(\text{ int })(M+1) = ((\text{ int })\&\underbrace{*M}_{M[0]}) + 2 \texttt{ * sizeof (float)}$$
Le tableau de 2 cases juste après la fin de$M[0]$
$$(\text{ int })(M[0]+1) = ((\text{ int })\&\underbrace{*M[0]}_{M[0][0]}) + \texttt{ sizeof (float)}$$
La case float juste après $M[0][0]$

Soit, sur un schéma :



L'utilisation des crochets marche un peu par miracle, grâce à l'arithmétique des pointeurs, car le programme sait toujours (grâce au type `matrice`) quelle est la taille des tableaux. On va utiliser des **pointeurs**, c'est-à-dire des variables contenant des adresses, pour *simuler* ce fonctionnement. On veut continuer à avoir :

— $M[i]$ est l'adresse de la ligne i ;
— $M[i][j]$ est l'élément j de la ligne i de la matrice.

Comme on ne peut plus compter sur la taille des lignes dans l'arithmétique des pointeurs, on va :

— Dire que M est un tableau contenant des adresses de lignes ;
Les éléments de M sont de type `float *`, et le type de M, comme adresse de la première case du tableau, est `float **`.
— Chaque ligne contient des `float`s
Les éléments d'une ligne $M[i]$ sont de type `float`, et le type de $M[i]$, comme adresse de la première case de la ligne, est `float *`.

Enfin, comme le nombre de lignes et de colonnes n'est plus le même pour toutes les matrices, on va utiliser une structure contenant à la fois les éléments de la matrice et son nombre de lignes et de colonnes :

```
struct matrice_s {
  int nb_lignes ;
  int nb_colonnes ;
  float ** m ;
} ;
```

(a) Créez un fichier `../../include/matrices.h` contenant :
— la déclaration d'un type `matrice` et de la structure `matrice_s` comme ci-dessus ;
— les déclarations de deux fonctions, `lire_matrice` et `afficher_matrice`.

Pour chaque des questions suivantes, écrivez chaque fonction dans un seul fichier.

(b) Écrivez une fonction `creer_matrice` qui prend en entrée un nombre de lignes et un nombre de colonnes et renvoi l'adresse d'une **structure** matrice_s ayant le bon nombre de lignes et de colonnes.

Solution

```
#include <malloc.h>

matrice creer_matrice ( int nb_lignes , int nb_colonnes )
{
  int i ;
  matrice res ;
  res = malloc ( sizeof ( struct matrice_s ) ) ;
  res->nb_lignes = nb_lignes ;
  res->nb_colonnes = nb_colonnes ;
  res ->m = ( float ** ) malloc ( res->nb_lignes * sizeof ( float * ) ) ;
  for ( i = 0 ; i < res->nb_lignes ; i ++ )
    res->m[i] = ( float * ) malloc ( res->nb_colonnes * sizeof ( float ) )
  return res ;
}
```

(c) Comme dans l'exercice précédent, écrivez les fonctions `lire_matrice` et `afficher_matrice`.

Solution

```
#include "entrees.h"
#include "matrices.h"

void lire_matrice ( matrice a  )
{
  int i , j ;
  for ( i = 0 ; i < a->nb_lignes; i++ )
  {
    printf ( "Coefficients␣de␣la␣ligne␣%d :\n" , i + 1 ) ;
    for ( j = 0 ; j < a->nb_colonnes ; j++ )
    {
      printf ( "M[%d][%d]␣=␣" , i + 1 , j + 1 ) ;
      LireDecimal ( ( M->m[i] ) +j ) ;
    }
  }
}


void affiche_matrice ( matrice M )
{
  int i , j ;
  for ( i = 0 ; i < M->nb_lignes; i++ )
  {
    printf ( "(\t%f\t" , M->m[i][0] ) ;
    for ( j = 1 ; j < M->nb_colonnes ; j++ )
      printf ( "%f\t" , M->m[i][j] ) ;
    printf ( ")\n''␣;
␣␣}
␣␣printf␣(␣"\n"␣)␣;
}
␣␣
```

(d) Ajoutez la règle suivante au fichier `Makefile` :

```
creer_matrice.o: ../../include/matrices.h creer_matrice.c
	gcc -Wall -c -o creer_matrice.o creer_matrice.c -I ../../include

lire_matrice.o: ../../include/matrices.h lire_matrice.c
	gcc -Wall -c -o lire_matrice.o lire_matrice.c -I ../../include

 afficher_matrice.o: ../../include/matrices.h afficher_matrice.c
	gcc -Wall -c -o afficher_matrice.o afficher_matrice.c -I ../../include

../../lib/libmatrices.a: creer_matrice.o  lire_matrice.o afficher_matrice.o
	ar rcs $@ $<
```

Explications : dans une règle, la variable `$@` est le but de la règles (qui est avant les deux points, donc `../../lib/libmatrices.a` ici), et la variable `$<` contient tout de qui est après les deux points, donc `creer_matrice.o lire_matrice.o afficher_matrice.o` ici.

(e) Écrivez dans un fichier `test_matrice.c` une fonction `main` qui va tester les fonctions de la librairie `libmatrices.a`. Écrivez dans le fichier `Makefile` une règle permettant de compiler ce programme. **Attention :** vous aurez aussi besoin de la libraire sur les entrées (pour la fonction `lire_matrice`. Vérifiez que tout marche bien avant de continuer.

(f) Étendez la libraire des matrices avec deux fonctions faisant la somme et le produit de matrices. **Ces fonctions doivent tester s'il est possible de faire ces opérations** en regardant les nombres de lignes et de colonnes de leurs argument. Leur premier argument est l'adresse d'une matrice (l'adresse de l'adresse d'une structure) qu'elles devront créer avec la bonne taille. N'oubliez pas de mettre à jour le fichier `Makefile` et de tester ces fonctions !

Solution

```
#include "matrices.h"
int somme_matrices ( matrice * res , matrice a , matrice b )
{
  int i , j ;
  if ( ( a->nb_lignes != b->nb_lignes ) || \
       ( a->nb_colonnes != b->nb_colonnes ) )
  {
    *res = NULL ;
    return 1 ;
  }
  *res = creer_matrice ( a->nb_lignes , a->nb_colonnes ) ;
  for ( i = 0 ; i < a->nb_lignes; i++ )
    for ( j = 0 ; j < b->nb_colonnes ; j++ )
      (*res)->m[i][j] = a->m[i][j] + b->m [i][j] ;
  return 0 ;
}


int produit_matrices ( matrice * res , matrice a , matrice b )
{
  int i , j , k ;
  if ( ( a->nb_colonnes != b->nb_lignes ) )
  {
    *res = NULL ;
    return 1 ;
  }
  *res = creer_matrice ( a->nb_lignes , b->nb_colonnes ) ;
  for ( i = 0 ; i < a->nb_lignes; i++ )
    for ( j = 0 ; j < b->nb_colonnes ; j++ )
    {
      (*res)->m[i][j] = 0 ;
      for ( k = 0 ; k < a->nb_colonnes ; k++ )
        (*res)->m[i][j] += a->m[i][j] * b->m [i][j] ;
```

```
    }
  return 0 ;
}
```